UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA
Departamento de Sistemas Informáticos y Computación

TESIS DOCTORAL

Desarrollo dirigido por modelos de aplicaciones seguras para el manejo de información

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Miguel Ángel García de Dios

Director

Manuel García Clavel

Madrid, 2015

Desarrollo Dirigido por Modelos de Aplicaciones Seguras para el Manejo de Información

TESIS DOCTORAL

Memoria presentada para obtener el grado de
Doctor en Informática

Miguel A. García de Dios

Dirigida por el profesor
Manuel García Clavel

Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

2015

Model-driven Development of Secure Data-Management Applications

PhD Thesis

Miguel A. García de Dios

Advisor
Manuel García Clavel

Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

2015

Acknowledgments

I am finishing my thesis at the age of 31. Throughout all this time I have shared my life with many people. I would like to take the opportunity to express my gratitude to all of them. Family, friends, girlfriends, colleagues: thanks to all of you for being part of my life all this years. I would not be here without you.

It would be far too long to mention each and every one of the persons to whom I am grateful. Therefore, I will dedicate the following words to the people that have been more relevant in my life at both personal and professional level.

At personal level, I want to thank my mother first, María Victoria, for everything that she has done for me. She has taught me how to be a good person, to act always in the most appropriate manner, to fight for my dreams, and to live happily whatever happens. She has taught me all that in the best possible way: showing me, day after day, how to overcome all the problems with a smile. This work is dedicated specially to her.

Thanks to my uncles and aunts for having been there, specially in the most difficult moments. I want to thank my aunt Ana for all the affection and support given during these years. She is like a mother to me.

Thanks to my cousin José. I have grown with him. We have always been together. I love him unconditionally.

I consider myself very lucky for having so many friends, and I want to thank all of them for being part of my life. But I want to thank my friend Juan (el Mono) in a special way for all the moments that we have shared and lived. I love him like a brother.

I want to thank Raquel for all the years we spent together. She made me happy and taught me many things. We shared very good times. I am grateful for all the support she always gave to me, and for all she did for my mother and for me. She has been one of the most influential person of my life.

Thanks to Gloria because, day after day, she makes me happy. I feel very lucky for sharing my life with her, for the way she is, for all she does for me, and for all the help she gives me everyday. With her, my life is simply much better.

I have many nice memories of Antonio Gistau residence hall. Thanks to all the staff, mates, and friends from there. They took care of me and stayed by my side. Particularly, I want to thank one of my best friends, Juan (el Chino), for always being there. I have shared an important part of my life with him and together, we have spent many good times. Even though we don't see each other very often we both know that we can count on each other for everything. Furthermore, although it has been a long time since we last met, I want to thank Joaquín (el Sote) for everything he taught me and for taking good care of me. He is a wise man whose advice was always very helpful.

I want to thank all my friends for accompanying me during the university period. Specially, I thank Carlos (el tito Charlie) and Ana. I love them for how they are and for everything we have shared together.

Thanks to Paqui and her daughter, Virginia, for everything they have done for my mother in the most difficult moments. With them, my life has been much better.

I also want to thank Karen for making me feel like home, during the six-months period that I spent in Zurich. The warmth of her home compensated (by far) that cold winter of 2013.

At professional level, first I want to thank Manuel for everything he has done for me. Apart from being my doctoral supervisor, he has taught me, helped me, advised me, supported me, and trusted me, at professional and personal levels, since the beginning. He is the main reason why I am here today.

I might also say the same about Marina. From the first day, she has trusted me and she has always helped my with everything I have needed. Her support and her advice have always been very helpful. Thank you.

I want to thank Carolina, my friend through thick and thin, for being there every time I have needed her, and for all her help. Carolina is always the first one to give a hand, no matter what the problem is. She has been a really important support throughout all these years, in every way.

I also want to thank Gonzalo and Javier for the time we spent together and for all their help.

Thanks to Narciso for all his patient and dedication.

I want to thank David for the wonderful six-months that I spent in Zurich. Living that experience was really positive and enriching. Thank you for your dedication.

I want to thank Imdea Software's personnel. To Manuel H., Manuel Ca. (Los Manueles), and María (the boss) for giving me the opportunity to achieve the PhD.

Thanks to the technicians Juan, Rober, and Gabriel for all the help and knowledge provided.

And thanks to the administration staff: Paola, Tania, Carlota, Andrea, and Laura. You make Imdea Software a better and funnier workplace.

Finally, thanks to all my relatives and friends that are not here anymore. They also have been an important part of my life. To all of you, thank you, part of this work belongs to you.

This research has been supported by the EU FP7-ICT Project "NESSoS: Network of Excellence on Engineering Secure Future Internet Software Services and Systems" (256980), by the Spanish Ministry of Science and Innovation Projects "DESAFIOS" (TIN2006-15660-C02-01) and "DESAFIOS-10" (TIN2009-14599-C03-01), by the Spanish Ministry of Economy and Competitiveness Project "StrongSoft" (TIN2012-39391-C04-04), by Comunidad de Madrid Program "PROMETIDOS-CM" (S2009TIC-1465), and by the Fundación Vodafone's one-year fellowship.

Agradecimientos

He terminado esta tesis doctoral con 31 años. Durante todo este tiempo he compartido mi vida con muchas personas a las cuales dedico estas líneas como forma de agradecimiento. Familiares, amigos, parejas, compañeros de trabajo: gracias a todos por haber formado parte de mi vida. Sin vosotros ahora mismo no estaría aquí.

Sería demasiado extenso nombrar a todas y cada una de las personas a las que estoy agradecido. Por ello, dedicaré las siguientes líneas a aquellas que han sido más relevantes en mi vida, tanto a nivel personal como a nivel profesional.

A nivel personal, primero quiero agradecer a mi madre, María Victoria, todo lo que ha hecho por mí. Ella me ha enseñado a ser buena persona, a actuar siempre de la mejor manera, a luchar por lo que uno quiere y a ser feliz en la vida, pase lo que pase. Y me lo ha enseñado de la mejor forma posible: mostrándome, día a día, cómo superar todos los problemas, siempre con una sonrisa. Este trabajo va dedicado especialmente a ella.

Agradezco a todos mis tíos el haber estado siempre ahí, sobre todo en los momentos más difíciles. En especial, agradezco a mi tía Ana todo el cariño y el apoyo que siempre me ha dado. Ella es mi segunda madre.

Le doy las gracias a mi primo José. Con él he crecido y mi vida siempre ha estado ligada a la suya, siempre a mi lado. A él le quiero de manera incondicional, es parte de mí.

Me considero muy afortunado por la gran cantidad de amigos que tengo, y les doy las gracias a todos ellos por formar parte de mi vida. Pero si hay uno con el que más cosas comparto, más confianza tengo y mejor me lo paso, ése es mi amigo Juán (el Mono). A él le quiero como a un hermano y ,por todo lo que significa para mí, le doy las gracias.

Quiero agradecer a Raquel todos los años que pasamos juntos. Ella me hizo feliz, me enseñó muchas cosas y con ella compartí muy buenos momentos. Le agradezco todo el apoyo que siempre me dio y todo lo que hizo por mi madre y por mí. Ella ha sido una de las personas más influyentes e importantes de mi vida.

Agradezco a Gloria que, día a día, me haga feliz. Me siento muy afortunado de compartir mi vida con ella, de su forma de ser y de todo lo que me ayuda y hace por mí. Con ella mi vida simplemente es mucho mejor.

Le doy las gracias a todos los trabajadores y compañeros de la Residencia Antonio Gistau. Ellos me cuidaron y me acompañaron durante mi etapa como estudiante. De todos ellos quiero agradecer especialmente a uno de mis mejores amigos , Juan (el Chino), el haber estado siempre a mi lado. Con él he compartido gran parte de mi vida y juntos hemos vivido infinidad de momentos. Aunque, por circunstancias de la vida, últimamente no nos veamos mucho, ambos sabemos que podemos contar el uno con el otro, siempre, para lo que sea. Además, y aunque haga mucho tiempo que no nos vemos, agradezco a Joaquín (el Sote) todo lo que me enseñó y lo que cuidó de mí. Un hombre muy sabio cuyos consejos siempre me han sido de gran ayuda.

Quiero agradecer a todos los amigos que me acompañaron durante la universidad. En especial le doy las gracias a Carlos (el tito Charlie) y a Ana. A ellos les tengo un cariño especial por cómo son y por todo lo que hemos vivimos juntos.

Agradezco a Paqui y a su hija Virgina todo lo que han hecho por mí y por mi madre en los momentos difíciles. Gracias a ellas mi vida ha sido mucho mejor.

También quiero agradecer a Karen el hacerme sentir como en casa los seis meses que pasé en Zúrich. El calor con el que me acogió y me trató en su casa compensó (con creces) aquel frío invierno del 2013.

A nivel profesional, primero quiero agradecer a Manuel todo lo que ha hecho por mí. Al margen de ser el director de mi tesis, él me ha enseñado, me ha ayudado, me ha aconsejado, me ha apoyado y ha confiado en mí, tanto a nivel personal como profesional, desde el primer día. Él es el principal responsable de que hoy esté aquí.

Lo mismo puedo decir de Marina. Desde el primer día ha confiado en mí y siempre me ha ayudado en todo lo que he necesitado. Su apoyo y sus consejos me han sido siempre de una gran ayuda. Gracias.

Agradezco a Carolina, mi compañera de batallas, por estar siempre ahí cuando lo he necesitado y por todo lo que me ha ayudado. En los momentos de crisis (laboralmente hablando), Carolina siempre ha sido la primera en ofrecerse a echar una mano, siempre "al pie del cañón". Ha sido un gran apoyo durante todos estos años, tanto en el plano profesional, como en el personal.

También agradezco, a Gonzalo y a Javier, el tiempo que compartimos juntos y todo lo que me ayudaron.

Agradezco a Narciso toda la paciencia y dedicación que ha tenido siempre hacia mí.

Gracias a David por permitirnos disfrutar de esa maravillosa estancia de seis meses en Zúrich. Vivir esa experiencia fue muy positivo y enriquecedor para mí. Gracias también por su trabajo y dedicación.

Quiero dar las gracias al personal de Imdea Software. Gracias a Manuel H., Manuel Ca. (Los Manueles) y a María (la jefa) por darme la oportunidad de poder realizar el doctorado.

Gracias a los técnicos Juan, Roberto y Gabriel por toda la ayuda y sabiduría proporcionada.

Y gracias a las chicas de administración Paola, Tania, Carlota Andrea y Laura por hacer que todo en Imdea sea mucho más fácil (y divertido).

Por último, agradezco a todos los familiares y amigos que ya no están con nosotros, pero que no por ello han sido menos importantes en mi vida. A todos vosotros y a todos ellos: gracias. Parte de este trabajo os pertenece.

Esta investigación ha sido apoyada por el Proyecto EU FP7-ICT "NESSoS: Red de Excelencia sobre la creación de Servicios y Sistemas de Software Seguros para el Internet del Futuro" (265980), por los Proyectos del Ministerio Español de Ciencia e Innovación "DESAFIOS" (TIN2006-15660-C02-01) y "DESAFIOS-10" (TIN2009-14599-C03-01), por el Proyecto del Ministerio Español de Economía y Competitividad "StrongSoft" (TIN2012-39391-C04-04), por el Programa de la Comunidad de Madrid "PROMETIDOS-CM" (S2009TIC-1465) y por la beca de investigación de un año de la Fundación Vodafone.

Abstract

We present a novel, tool-supported model-driven methodology for developing secure data-management applications. Data-management applications are focused around so-called CRUD actions that create, read, update, and delete data from persistent storage. These operations are the building blocks for numerous applications, for example dynamic websites where users create accounts, store and update information, and receive customized views based on their stored data. When the data managed is sensitive, then security is a concern and the use of these actions must be controlled.

Within our methodology, developers proceed by modeling three different views of the desired application: its data model, security model, and GUI model. These models formalize respectively the application's data domain, authorization policy, and its graphical interface together with its behavior. Afterwards a model-transformation function automatically lifts the policy specified by the security model to the GUI model. This allows a separation of concerns where behavior and security are specified separately, and subsequently combined to generate a security-aware GUI model. Finally, a code generator automatically generates a multi-tier application, along with all support for access control, from the security-aware GUI model.

Overall, we see our contributions as follows. First, our methodology offers Model Driven Architecture's purported benefits for data-management applications. By working with models, developers can focus on the application's data, behavior, security, and presentation, independent of the different, often complex, technologies that are used to implement them. Second, our use of model transformations leads to modularity and separation of concerns: the GUI model and the security model can be changed independently and by different developers, if desired. This avoids the problems with hardcoded security policies that are difficult to maintain and audit. Finally, our methodology is quite powerful and compares favorably to alternatives. In particular, it leverages well-known security languages for modeling rich, fine-grained access control policies, which must often be manually encoded in other proposals. Moreover, our new language for GUIs supports modeling realistic, dynamic web interfaces (where the web content varies based on the user's actions or user-provided data), without limiting the interfaces to a fixed set of templates or interaction patterns, as in other methodologies. Of course, the proof of the pudding is in the eating and we report on applications that we developed, which provide evidence of the applicability of this approach.

Resumen

En esta memoria presentamos una metodología original encuadrada en el área de desarrollo de software dirigido por modelos. En particular, es una metodología automática apoyada en herramientas para el desarrollo de aplicaciones seguras de gestión de datos guardados en una capa persistente. Las aplicaciones de gestión de datos se centran en las llamadas acciones CRUD de crear, leer, modificar y borrar datos del almacenamiento persistente. Estas operaciones son el bloque básico de numerosas aplicaciones como, por ejemplo, páginas web donde los usuarios crean cuentas, almacenan y modifican información y reciben vistas personalizadas basadas en los datos que almacenan. Cuando los datos gestionados son sensibles, medidas de seguridad deben protegerlos, por lo que el uso de estas acciones debe ser controlado.

En nuestra metodología, los desarrolladores proceden modelando tres vistas de la aplicación deseada: el modelo de datos, el modelo de seguridad y el modelo de la interfaz gráfica de usuario (GUI, de sus siglas en inglés). Estos modelos formalizan respectivamente un dominio de datos, una política de autorización y una interfaz gráfica junto con los eventos que permiten la interacción con el usuario y dirigen el flujo de información. Después, una función de transformación de modelos traslada automáticamente la política especificada por el modelo de seguridad al modelo de GUI. La metodología descrita permite la separación de tareas, de manera que el comportamiento y la seguridad se especifican de forma separada y posteriormente se combinan para generar un modelo de GUI seguro. Finalmente, un componente de generación de código genera de forma automática una aplicación multicapa, junto con todo el soporte de control de acceso, a partir del modelo de GUI con seguridad.

A continuación describimos las que son, en nuestra opinión, las contribuciones principales. Primero, nuestra metodología ofrece los beneficios pretendidos por la Arquitectura Dirigida por Modelos (MDA, por sus siglas en inglés) para las aplicaciones de gestión de datos. Trabajando con modelos, los desarrolladores pueden centrarse en la estructura de los datos, el comportamiento, la política de seguridad de la aplicación y su presentación, independientemente de las diferentes (y a menudo complejas) tecnologías que son usadas para implementarlas. Segundo, nuestro uso de transformaciones de modelos hace posible la modularidad y la separación de aspectos: el modelo de GUI y el modelo de seguridad pueden modificarse de manera independiente y por diferentes desarrolladores, si se desea. Esto evita los problemas relacionados con modificar a mano las políticas de seguridad, las cuales son difíciles de mantener y analizar. Finalmente, nuestra metodología es realmente potente, comparándose favorablemente con otras alternativas. En particular, toma ventaja de los bien conocidos lenguajes de seguridad para modelar políticas de control de acceso ricas y de grano fino, las cuales tienen que ser a menudo codificadas manualmente en otras propuestas. Además, nuestro nuevo lenguaje para GUIs soporta el modelado de interfaces web dinámicas para la gestión de datos reales (donde el contenido web varía según las acciones del usuario), sin limitar las interfaces a un conjunto fijo de plantillas o patrones de interacción como en otras metodologías. Por supuesto, no se puede evaluar el éxito de esta metodología sin haberla llevado a la práctica, por lo tanto, detallamos información sobre aplicaciones que hemos desarrollado con nuestra metodología, las cuales evidencian la aplicabilidad de la misma.

Contents

List of Figures

Part I

Summary of the research

Chapter 1

Introduction

Model building is at the heart of system design. This is true in many engineering disciplines and is increasingly the case in software engineering. Proponents of model-driven engineering have in the past been guilty of making overambitious claims: positioning it as the Holy Grail of software engineering where modeling completely replaces programming. However, there are specialized domains where MDE can truly deliver its full potential: in our opinion, security-aware GUIs for data-centric applications is one of them.

1.1 Model-driven software engineering

The ever-growing development and use of information and communication technology is a constant source of security and reliability problems. Clearly we need better ways of developing software systems. Model-Driven Engineering (MDE) [61] is a software development methodology that focuses on creating models of different system views from which system artifacts such as code and configuration data are automatically generated.

Domain-specific modeling languages In our opinion, however, only by limiting the domain it is possible to define sufficiently precise modeling languages that support the automatic generation of fully functional applications. Arguably, the late adoption of MDE is due to the difficulty in defining effective domain-specific modeling languages and also to the effort required for developers to learn modeling languages and the art of model building. Defining a good domain-specific modeling language requires finding the right abstractions and degree of precision to capture relevant aspects of the structure and the logic of a software system. Moreover, for a modeling language to be really usable and useful for software developers, appropriate tools must be provided to build models, analyze them, and keep them synchronized with end products.

Model-transformations In MDE, transformation is the way of using models to produce other development artifacts. The following types of transformations have been widely used so far:

- *Generation of code and execution artifacts:* Models may be mapped to code or other artifacts that affect the system's runtime behavior. When generating code, the transformation function amounts to a kind of translator or compiler. Examples of other artifacts that can be generated from models are deployment and configuration data, which also affects the system's behavior.

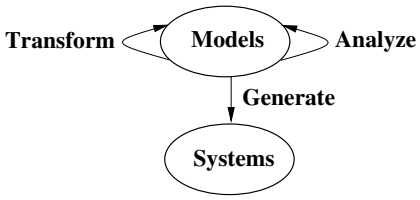


Figure 1.1: Use of models in model-driven security

- *Generation of models:* Models may be mapped to other models. Typically such transformations add details, specialize constructs, or change representations. An example of this is the specialization of platform-independent models to platform-specific models. In general, model transformations support problem decomposition during development where design aspects can be separated into different models which are later composed.
- *Generation of test cases:* Test cases can be generated from models.

1.2 Model-driven security

Model-driven security[66, 11, 7, 8, 36, 9, 44] is a specialization of MDE to the domain of security. As discussed in [9], models can be used for the following four activities in the development of secure systems:

A1. Precisely documenting security requirements together with design requirements.

A2. Analyzing security requirements.

A3. Model-based transformation, such as migrating security policies on application data to policies for other system layers or artifacts.

A4. Generating code, including complete, configured security infrastructures.

Figure 1.1 depicts these activities and their interrelationships. Designers specify security-design models that combine security and design requirements (A1). When modeling languages have a well-defined semantics, one can formally analyze these designs (A2). When designing secure systems, security may be relevant at different system layers or views. Using model transformations, one can migrate a security policy from one model to other models (A3). Finally, one can use tools to automatically generate code and other system artifacts directly from the models (A4).

The crucial part of the specialization that model-driven security brings about concerns the modeling language. Instead of adopting a one-language-fits-all approach, [11] proposes a general schema for integrating security requirements into system design models. The main idea is to define security modeling languages that are general in that they leave open the nature of the protected resources, i.e., whether these resources are data, business objects, processes, controller states, etc. Figure 1.2 provides examples of different security notions which could be specified using a security modeling language (top-left) that one might integrate with different design modeling languages (bottom-left), resulting in a security-design modeling language (right side). For example, one might combine a modeling language for Role Base Access Control (RBAC) with Class Diagrams, as indicated in bold in the figure. This combination is made by defining a dialect (or "glue"), which identifies elements of the design language as the protected resources of the security

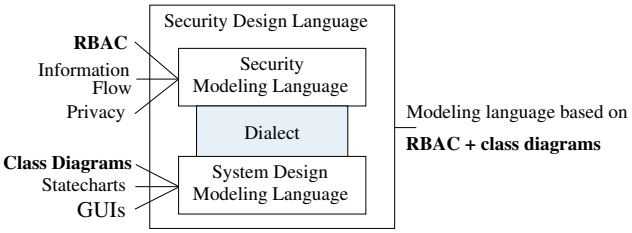


Figure 1.2: Model-languages and their combination

language. In this way, one can flexibly define languages for formulating different kinds of system designs along with their security requirements.

1.3 Secure data-management applications

Data-management applications are focused around so-called CRUD [1] actions that create, read, update, and delete data from persistent storage. These operations are the building blocks for numerous applications, for example dynamic websites where users create accounts, store and update information, and receive customized views based on their stored data. When the data managed is sensitive, then security is a concern and the use of these actions must be controlled.

Access control is the standard approach to restricting users' actions on data. When the access-control policies are sufficiently simple, it may be possible to formalize them declaratively, independent of the application's business logic. For example, multi-tier systems for web-based applications often build support for role-based access control into the application server, which is configured independently of the application's procedural details. In contrast, fine-grained access control policies may depend not only on the user's credentials but also on the satisfaction of constraints on the state of the persistence tier, i.e. on the values of stored data items. In such cases, authorization checks are typically implemented programmatically, by directly encoding them at appropriate places in the application. Unfortunately, these programmatic additions are cumbersome, error prone, and scale poorly. Moreover, they are difficult to audit and maintain as the authorization checks are spread throughout the code, and security policy changes require code changes.

1.4 Model-driven development of secure data-management applications

We propose in this work a methodology for the model-driven development of secure data-management applications. It consists of languages for modeling multi-tier systems, and a toolkit for generating these systems. Within our methodology, a secure data-management application is modeled using three interrelated models:

1. A *data model* defines the application's data domain in terms of its classes, attributes, associations, and (side-effect free) methods;

2. A *security model* defines the application's security policy in terms of authorized access to the actions on the resources provided by the data model.

[1] This is true for applications in which their persistent layer is a database. When the persistent layer is either a LDAP or cloud storage, the actions are not exactly the same.

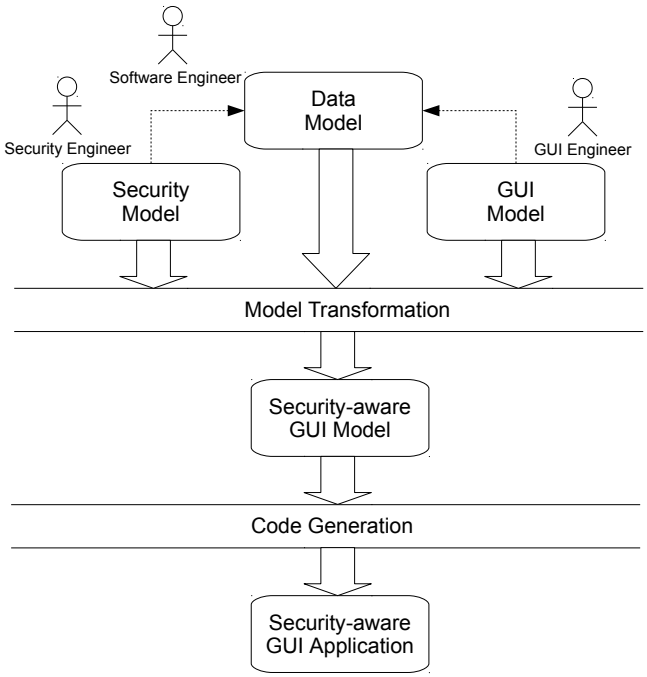


Figure 1.3: Model-driven development of security-aware GUIs.

3. A graphical user interface, or *GUI model*, defines the application's graphical interface and application logic. Note, in particular, that this model formalizes both *UI structure* and *behavior*.

The heart of this methodology, illustrated in Fig. 1.3, is a model-transformation function that automatically lifts the policy that is specified in the security model to the GUI model. The idea is simple but powerful. The security model specifies under what conditions actions on data are authorized. The control information in the GUI model specifies which actions are executed in response to which events. Lifting essentially consists of prefixing each data action in the GUI model with the authorization check specified in the security model. The resulting GUI model is security aware. It specifies UI structure, information flow with persistent storage, and all authorization checks.

We have implemented this methodology within a toolkit, called ActionGUI [1], that performs this many-models-to-model transformation. From the resulting security-aware GUI model, ActionGUI generates a deployable application, along with all support for access control, based on the following, standard three-tier architecture.

1. Presentation tier: Users access web applications through standard web browsers, which render the content (HTML and JavaScript) dynamically provided by the web server.

2. Application tier: The toolkit generates Java Web Applications. The applications run in a web server (such as Tomcat or GlassFish), processing client requests and generating content, which is sent back to the client for rendering. When processing client requests, the generated applications *interpret* their underlying security-aware GUI models.

3. Persistence tier: The generated application manages information stored in a database. For each application, the toolkit generates the corresponding database schema from

the application's data model.

The author's main contributions

ActionGUI is the latest result of a very ambitious project to which several researchers have contributed in different ways over the past seven years.

The beginnings of the ActionGUI project can be traced back to 2008, and in particular to the project reported in [36]. The main driver behind this seminal work was the use of model-driven security for developing an industrial application. The authors succeeded in modeling, using SecureUML [11], the application's fine-grained access control policy. However, the process of manually encoding the modeled policy within the application's graphical user interface was unreasonably time-consuming and error-prone.

This unsatisfactory state of affairs provided the *raison d'être* for the ActionGUI project: namely, the quest for

- *a model-driven solution for lifting the access control policy to the application's graphical user interface.*

Furthermore, it provided the project's *success criteria*: namely, that the intended solution should

- *support full model-driven development (from models to code)* and
- *be applicable in industrial-strength applications.*

As detailed below, the author's contributions to the ActionGUI project has been always directed towards the satisfaction of these success criteria. In fact, one can say in all fairness that he is responsible of the extent to which these success criteria are currently satisfied.

The first concrete model-driven solution to the aforementioned problem came in 2009. In [77, 76, 13] the idea of using a many-models-to-model transformation to make a GUI model *security-aware* with respect to a security model was introduced for the first time. This solution was soon supported by SSG [44], a novel development environment for automatically building security-aware GUIs.

SSG was in fact the author's first contribution to the ActionGUI project. It consisted of a number of Eclipse plugins including three model editors, a model-transformation tool, and a code generator. A key part of the SSG's seminal code generator was EOS [33], a Java component designed and implemented by the author for efficiently evaluating OCL expressions. The author also implemented a mapping from OCL to first-order logic for checking the unsatisfiability of OCL constraints [34] using SMT solvers. This mapping was further elaborated in [39] and is currently integrated in the ActionGUI toolkit. Among other applications, it has been successfully used to formally reason about fine-grained access control policies [43].

According to the ActionGUI project's success criteria, it was necessary to test the proposed model-driven solution and toolkit in a real scenario. To this end, the author obtained from the Fundación Vodafone a one-year grant (September 2010 – August 2011) for developing a CRM solution and a volunteer management system for a Hospital and Care Center in Madrid.

To carry out these applications, which were eventually reported in [12], the author had to extend in many decisive ways the solution originally proposed in [13] as well as its associated toolkit [44]. In particular, at the level of the modeling language for GUIs, the author introduced *widget variables* as a key modeling element, and *tables* and *combo-boxes* as new widget types. Then, at the level of the modeling language for security policies,

the author extended SecureUML in two crucial ways: first, by replacing the action of updating a link between two objects (whose meaning was originally ambiguous) with two new separate actions, one for *creating* a link and the other for *deleting* it; and, secondly, by allowing two new special variables, namely value and target, to appear within authorization constraints, in order to significantly increase its expressiveness. The author, as the main responsible at that time of the ActionGUI toolkit, was the one in charge of implementing, all these new modeling features. Our enriched solution to the problem of making a GUI model security-aware with respect to a security model was first presented in [10], along with the associated toolkit.

The next major case study was carried out by the author in the context of the NESSoS project (October 2010 –March 2014). NESSoS is the European Network of Excellence on Engineering Secure Future Internet Software Services and Systems [71]. The case study, which is reported in [42], consists of a web-based system for electronic health record management. While carrying out this case study, the author discovered a major limitation in the model-transformation for lifting the security policy to the GUI level. Let us briefly discussed this limitation and the successful solution eventually proposed by the author.

In the previous work [13, 10] the lifting of the security policy to the GUI level consisted of prefixing each event in the GUI model with the authorization check specified in the security model. However the author realized that, since the data actions executed by an event may change the persistence tier's state, checking authorization at the level of events, and therefore before executing any data action, is sufficient only (i) if the underlying security policy does not contain authorization constraints, or (ii) if they do not depend on values that are changed during the execution of the event's data actions. To overcome this limitation, the author proposed to check authorizations before executing each event's data action, while providing at the same time events with a *transaction* semantics: either all of the data actions are executed in the given order, or none of them are executed at all. This substantial generalization of [13, 10] appeared first in [12, 6] and is the one currently supported in the ActionGUI toolkit.

Last but not least, the author has significantly contributed to the ActionGUI project by writing the user manual and the installation manual of the ActionGUI toolkit, as well as the technical documentation of the ActionGUI modeling languages. It is worth noticing that this material (in its different versions) has been successfully used by students and instructors in various courses, including: "Automatic Generation of Smart, Security-Aware GUI Models." (Seminar on Model-Driven Engineering, IRISA, Rennes, France, November 2009); "Model-driven security: foundations, tools, and practice" (11th International School on Foundations of Security Analysis and Design, Bertinoro, Italy, September 2011); "Security Engineering" (Master-level course, ETH Zurich, Switzerland, October 2012 – January 2013); "ActionGUI Day" (Industrial training day, IMDEA Software, June 2013), and "Model-Driven Engineering in Action" (Faculty course, Industrial University of Ho Chi Minh City, Vietnam, January – February 2014). The latest versions of the ActionGUI user and installation manuals are available at the ActionGUI web site [1].

1.5 Summary

- Chapter 2: *A methodology for developing secure data-management applications.* We present ActionGUI as a novel model-driven methodology for developing secure data-management applications. System developers proceed by modeling three different views of the desired application: its data model, security model, and GUI model. These models formalize respectively the application's data domain, authorization

policy, and its graphical interface together with the application's behavior. Afterwards a model-transformation function lifts the policy specified by the security model to the GUI model. This allows a separation of concerns where behavior and security are specified separately, and subsequently combined to generate a security-aware GUI model.

- Chapter 3: *Supporting our methodology for developing secure data-management applications.* After a brief report on the current status of the ActionGUI toolkit, we report on our experience developing the Eye OCL Software (EOS) evaluator, a Java component for efficient OCL evaluation on medium-large scenarios, and we also explore various approaches for evaluating OCL expressions on really large scenarios. Then, we propose a mapping from a subset of OCL into first-order logic (FOL) and use this mapping for checking the unsatisfiability of sets of OCL constraints. We argue that our mapping is both simple, since the resulting FOL sentences closely mirror the original OCL constraints, and practical, since we can use automated reasoning tools, such as automated theorem provers and SMT solvers to automatically check the unsatisfiability of non-trivial sets of OCL constraints. Finally, we propose a methodology for reasoning about fine-grained access control policies, whose authorization constraints are specified in OCL, using the aforementioned mapping from OCL to FOL.

- Chapter 4: *Developing secure data-management applications with our methodology.* We provide a detailed report on our use of ActionGUI to develop a secure data-management application. This application is based on a case study proposed within NESSoS, the European Network of Excellence on Engineering Secure Future Internet Software Services and Systems [71]. The eHealth case study consists of a web-based system for electronic health record management (EHRM). Electronic health records (EHR) store information created by, or on behalf of, a health professional in the context of the care of a patient. The eHealth case study is interesting as an example of developing a secure data-management application and it provides a proof-of-concept for the application of the ActionGUI methodology to an industry-relevant problem. To provide further evidence of the usefulness of our methodology, we also include in this Chapter a summary of four other web applications that we have developed using ActionGUI.

- Chapter 5: *Related work.* We focus on works that are directly related to our main contribution, namely, ActionGUI as a novel, tool-supported model-driven methodology for developing secure data-management applications. First, we compare ActionGUI with other proposals for modeling data-management applications. Secondly, we compare ActionGUI with other tools (in this case, commercial tools) for developing secure data-management applications.

- Chapter 6: *Concluding remarks and future work.* We discuss two main directions along which we will further develop the ActionGUI project. First, we plan to enhance the ActionGUI toolkit so as to turn it into a full, robust, industrial-strength development platform. Secondly, we plan to extend the applicability of the ActionGUI technology by generalizing its main technological components and by implementing them as software services (SaaS).

Chapter 2

Modeling secure data-management applications

In this Chapter we present ActionGUI as a novel model-driven methodology for developing secure data-management applications. System developers proceed by modeling three different views of the desired application: its data model, security model, and GUI model. These models formalize respectively the application's data domain, authorization policy, and its graphical interface together with the application's behavior. Afterwards a model-transformation function lifts the policy specified by the security model to the GUI model. This allows a separation of concerns where behavior and security are specified separately, and subsequently combined to generate a security-aware GUI model.

In [10, 13, 44] we proposed the idea of using model transformations, to lift the security policy formulated in terms of the data model, to the GUI model. Here we improve and generalize this previous work. Lifting previously consisted of prefixing each event in the GUI model with the authorization check specified in the security model. However since the data actions executed by an event may change the persistence tier's state, checking authorizations at the level of events, and therefore before executing any data action, is sufficient only if the underlying security policy does not contain authorization constraints (which was explicitly assumed in [13]), or if they do not depend on values that are changed during the execution of the event's data actions (as was the case in the examples discussed in [10]). To overcome this limitation, we check now authorizations before executing each event's data action and we provide events with a *transaction* semantics: either all of the data actions are executed in the given order, or none of them are executed at all. The complete formal account of our methodology is given in the technical report [6]. Here we provide instead a high-level account of the correctness of the model-transformation function, which lies at our methodology's core.

2.1 Background

For modeling an application's data and security policy, we leverage existing modeling languages, namely, ComponentUML and SecureUML [11]. In this section we briefly introduce these languages. Since SecureUML uses the Object Constraint Language (OCL) [73] to model authorization constraints, we also summarize its main features.

2.1.1 ComponentUML

Data models provide a data-oriented view of a system. Typically they are used to specify how data is structured, the format of data items, and their logical organization, i.e., how data items are grouped and related. Our methodology employs ComponentUML for data modeling. ComponentUML provides a subset of UML class models where *entities* (classes) can be related by *associations* and may have *attributes* and *methods*. In ComponentUML, associations are binary: they always have two *association-ends* connecting two, not necessarily distinct, entities.

While ComponentUML and SecureUML have a graphical concrete syntax (see [11]), to simplify and clarify the presentation, we shall use textual concrete syntax. The complete definition of the ComponentUML syntax is given in Appendix A. In this syntax, entities are declared with the keyword **Entity** followed by the entity's name, and its attributes and association-ends, which are enclosed within brackets. Attributes and association-ends are declared together with their types. Moreover, since associations are binary, each association-end is declared together with its opposite association-end, designated by the keyword **oppositeTo**. Multiplicities other than ***** and **1** are specified using OCL invariants. Finally, comments are introduced with //.

As the following example illustrates, data models specify how the application's data is structured, independently of how it will be visualized or accessed.

Example 1 We use a simple chatroom application as a running example throughout this chapter. A demo version of this application can be found at [1]. The application provides an online discussion site where users converse by posting messages. Note that there are two types of users: registered and unregistered users. Registered users have their nicknames and passwords stored in the persistence tier. As usual, some options are only available to registered users, who log into the application by entering a valid nickname and password.

Here we use ComponentUML's textual syntax to model the chatroom's data model. The model, called ChatRoomDTM, consists of three entities representing chatrooms, registered users, and messages. The associations between these entities represent the relation between the registered users and the chatrooms in which they participate, the relation between the registered users and the messages that they have written, and the relation between the messages and the chatrooms where they have been posted. The entities' attributes represent that each chatroom has a topic, each chatroom can be public or not, each registered user has a nickname and a password, and each message has a body.

```
1  Entity Chatroom {
2    String topic
3    Boolean public
4    //registered users participating in this chatroom
5    Set(User) participants oppositeTo chatrooms
6    //messages posted in this chatroom
7    Set(Message) messages oppositeTo chatroom }

8  Entity User {
9    String nickname
10   String password
11   //chatrooms in which this registered user participates
12   Set(Chatroom) chatrooms oppositeTo participants
13   //messages written by this registered user
14   Set(Message) messages oppositeTo owner }
```

```
15 Entity Message {
16   String body
17   //chatroom where this message is posted
18   Chatroom chatroom oppositeTo messages
19   //registered user that wrote this message
20   User owner oppositeTo messages }
```

2.1.2 Object Constraint Language (OCL)

The Object Constraint Language (OCL) [73] is a language for specifying constraints and queries using a textual notation. As part of the UML standard, it was originally intended for modeling properties that could not be easily expressed using graphical notation, such as class invariants in a UML class diagram. Every OCL expression is written in the context of a model (called the *contextual model*), and is evaluated on an object model (also called the *instance* or *scenario*) of the contextual model. This evaluation returns a value but does not alter the given object model, since OCL's evaluation is side-effect free.

OCL is strongly typed. Expressions either have a primitive type, a class type, a tuple type, or a collection type. OCL provides standard operators on primitive data, tuples, and collections. For example, the operator →includes checks whether an object is part of a collection. OCL also provides a dot-operator to access the values of the objects' attributes and association-ends in the given scenario. For example, suppose that the contextual model includes a class c with an attribute at and an association-end as. Then, if o is an object of the class c in the given scenario, the expression $o.at$ refers to the value of the attribute at for the object o in this scenario, and $o.as$ refers to the objects linked to the object o through the association-end as. Finally, OCL provides operators to iterate over collections, such as →forAll, →exists, →select, →reject, →collect, and →iterate.

2.1.3 SecureUML

SecureUML [11] extends Role-Based Access Control (RBAC) [52] with *authorization constraints*. These constraints can be used to specify policies that depend on properties of the system state, for example, that a user can only post a message to a chatroom where the user participates. More specifically, SecureUML allows one to formalize access control decisions that depend on two kinds of information:

1. *static information*, namely the assignments of users and permissions to roles, and the role hierarchy, and

2. *dynamic information*, namely the satisfaction of authorization constraints in the current system state.

SecureUML therefore supports the modeling of *roles* and their hierarchies, *permissions*, *actions*, *resources*, and *authorization constraints*. Moreover, one can also model assignments: which permissions are assigned to a role, which actions are allowed by a permission, which resources are affected by a permission, and which authorization constraint must be satisfied before granting a permission.

SecureUML is, however, generic in that it leaves open the nature of the protected resources, i.e., whether these resources are data, business objects, processes, controller states, etc. In our methodology, we use an extension of SecureUML [11] that combines SecureUML with ComponentUML. In this extension, which for the sake of brevity we will

Table 2.1: SecureUML+ComponentUML: actions and resources.

Resource	Atomic Actions	Composite Actions
Entity	create, delete	read, update, full access
Attribute	read, update	full access
Method	execute	
Association-end	read, create, delete	full access

still call SecureUML, the protected *resources* are the entities, along with their attributes, methods, and association-ends, while the *actions* are those shown in Table 2.1.

Note that there are two classes of actions: atomic and composite. *Atomic actions* are intended to map directly onto existing operations on the persistence tier. *Composite actions* hierarchically group lower-level atomic actions. For example, the full access composite action for an attribute groups together the read and update atomic actions for this attribute. Finally, authorization constraints are specified using OCL, where the context of an OCL expression is the underlying data model. Additionally, OCL expressions in security models may contain the variables self, caller, value, and target, which are interpreted as follows:

- self refers to the resource upon which the action will be performed if the permission is granted. Notice that the resource of an attribute, a method, or an association-end is the entity to which it belongs.
- caller refers to the user that will perform the action if the permission is granted.
- value refers to the value that will be used to update an attribute if the permission is granted.
- target refers to the object that will be added (or removed) at an association-end if the permission is granted.

The reader familiar with the original presentation of SecureUML [11] may notice that we have introduced two new variables that can be used in authorization constraints: the variables value and target. Furthermore, to avoid potential ambiguities, we have refined the association-end update action into two separate actions: association-end create and association-end delete.

The complete definition of our SecureUML syntax is given in Appendix B. In this syntax, the entity modeling the system's users (or, more specifically, the system's *callers*) is declared with the keyword **User**. The roles that these users can take are declared with the keyword **Role** followed by the role's name, and its permissions, which are enclosed within brackets. The keyword **inherits**, appearing between two roles, declares that the first role is subordinated to the second role in the role hierarchy, and therefore inherits all its permissions.

Permissions are introduced by naming the resources to which they grant access. Each permission consists of a list of actions through which the corresponding root resource can be accessed. Actions on attributes, methods, or association-ends are declared along with their names. For example, **Read::***attr* denotes the read action on the attribute *attr*. The **if**–**then** construction is used to declare that the permission to execute an action is constrained by a condition. This condition is the authorization constraint that is associated to the permission.

As the following example illustrates, security models specify the application's access control policy in a fine-grained way. These models depend, of course, on how the application's data is structured, but not on how it is visualized or accessed through the application's graphical user interface.

Example 2 We use the SecureUML's textual syntax to model a policy for posting and reading chatroom messages. Our model, called ChatRoomSTM, has two roles: the role DefaultR represents everybody, i.e., both registered and unregistered users, and the role UserR represents only registered users. Our policy states that everybody can read any message posted in a public chatroom, but that only registered users can read messages posted in a private chatroom, provided they participate in that chatroom. Moreover, only registered users can post messages in public chatrooms; they can also post to private chatrooms, provided they also participate in that chatroom.

```
1  User User
2  Role DefaultR {
3    Chatroom {
4      //everybody can access the messages posted in a
5      //public chatroom
6      if self.public then Read::messages }
7    Message {
8      //everybody can read the body of any message
9      //posted in a public chatroom
10     if self.chatroom.public then Read::body } }

11 Role UserR inherits DefaultR {
12   Chatroom {
13     //every registered user can access the messages that
14     //are posted in a chatroom in which she participates
15     if self.participants→includes(caller)
16       then Read::messages }
17   Message {
18     //every registered user can read the body of any
19     //message that is posted
20     //in a chatroom in which she participates
21     if self.chatroom.participants→includes(caller)
22       then Read::body
23     //every registered user can create a new message
24     Create
25     //every registered user can claim ownership of any
26     //unowned message
27     if self.owner.oclIsUndefined() and target=caller
28       then Create::owner
29     //every registered user can change the body of any
30     //message she owns
31     //provided it is not yet posted anywhere
32     if self.owner=caller
33         and self.chatroom.oclIsUndefined()
34       then Update::body
35     //every registered user can post in a public chatroom any
36     //message she owns,
37     //provided it is not yet posted anywhere
38     if self.owner=caller and target.public
```

```
39          and self.chatroom.oclIsUndefined()
40        then Create::chatroom
41      //every registered user can post, in a chatroom in which
42      //she participates, any message she owns,
43      //provided it is not yet posted anywhere
44      if self.owner=caller and target.participants
45          →includes(caller)
46        and self.chatroom.oclIsUndefined()
47        then Create::chatroom } }
```

SecureUML provides various constructs for expressing complex access control policies compactly and intuitively, for example, by using action and role hierarchies or by declaring default policies. Nevertheless, as we will describe in Section 2.3.1, every security model S can be uniquely transformed into a semantically equivalent model $S^\flat$ for which the following holds:

Remark 1 *Let S be a security model. Then, for every atomic action act and every role r in S, there is exactly one permission in $S^\flat$ (possibly constrained by* false*) for r to execute act.*

Informally, the model $S^\flat$ makes the security policy specified in S completely explicit. Thus, let *Auth* be the function that, for every security model S, role r, and action *act*, returns the authorization constraint associated to the unique permission that is defined in $S^\flat$ for r to execute *act*. We will use this function *Auth* to define the model-transformation that, in our methodology, lifts the security policy from the security model to the GUI model. We conclude this section with some examples that illustrate in which sense *Auth* makes the security policy specified in a security model explicit.

Example 3 Consider the chatroom's security model, **ChatRoomSTM**, in Example 2. Note that UserR is a subrole of DefaultR (in line 11), which means that UserR will inherit all the DefaultR's permissions. Thus, Auth(**ChatRoomSTM**, UserR, **Read::**body) returns:

```
self.chatroom.public (from ln. 10)
or
self.chatroom.participants→includes(caller) (from ln. 21).
```

Note also that the association-end messages is opposite to the association-end owner. This means that a create (respectively delete) action on messages will be constrained by the same authorization that constrains a create (respectively delete) action on owner, having simultaneously replaced the variable self by target and the variable target by self. Thus, although no permission is explicitly given for the role UserR to execute a create action on messages, Auth(**ChatRoomSTM**, UserR, **Create::**messages) returns:

```
target.owner.oclIsUndefined() and self=caller (from ln. 27).
```

Finally, note that there is no permission explicitly given to the role DefaultR for executing an update action on the attribute body. Since permissions are denied by default (and no other rules can be applied in this case, like the ones for role inheritance or opposite association-ends) Auth(**ChatRoomSTM**, UserR, **Update::**body) returns false.

2.2 GUI Models

GUI models provide a human-interface oriented view of a system. Together with data models, they constitute platform independent application models, omitting security aspects.

Informally, a GUI consists of widgets, which are visual elements that display information and trigger events that execute actions. In this section we present a key component of our methodology: a novel language for modeling GUIs for data-management applications, called GUIML (GUI Modeling Language). It is important, however, to understand that GUIML is a language for modeling not only the *structure* of a GUI, i.e, the elements (widgets) that comprise it, but also the GUI's *behavior*, i.e., how its elements will react (actions) in response to user interactions with them (events). In fact, the key feature of GUIML is the language it provides for modeling the GUI's behavior, which uses OCL to specify both the conditions and the arguments for the different actions. The complete definition of the GUIML syntax is given in Appendix C. In this syntax, OCL expressions are enclosed in square brackets. This feature enables both the security model and the GUI model to "speak" the same language (namely, OCL in the context of the common, underlying data model). This allows us to define rigorously the transformation function that lifts the security policy to the GUI level.

We next briefly describe the main elements of GUIML, namely, *widgets* (with their associated *variables*), *events*, and *actions*. We will also illustrate them later with a simple example: a window for our chatroom application, where users can read and post messages in a chatroom.

Widgets

A GUI model consists of widgets of different types: windows (pages, when referring to web applications), combo-boxes (selectable lists), tables, date fields, boolean fields (check boxes), buttons, text fields, and labels. Widgets can be displayed in *containers*, which are also widgets. Widgets other than windows must be contained in another widget, and only windows, combo-boxes and tables may contain other widgets. Widgets may own variables, which store values for later use, and trigger events, which execute actions.

In concrete syntax, a widget is declared with a keyword like **Window**, **Button**, and **TextField**, according to its type, followed by the widget's (local) name, and the declaration of the variables it owns, the events it triggers, and the widgets it contains, all enclosed in brackets. The *global name* of each widget must be unique. If a widget is a window, its global name is the name given in its declaration. Otherwise, the global name results from concatenating, using dot, the global name of the widget's container with the name given in its declaration.

Variables

Each widget declaration may contain variable declarations, listing the variables owned by the widget. In concrete syntax, a variable declaration consists of the variable's type followed by its name.

There are also variables that are, by default, owned by every widget of a given type. These variables are implicitly declared in every widget declaration, and their values are handled in special ways. Here we only discuss the predefined variables that we will use in our example. The variables caller and role are predefined in every window. They store, respectively, the application's user and the user's role. The variable text is predefined in

every label, button, and text field. This variable stores the string displayed on the screen within the label, button, and text field; also, when a user types in a text field, the value of its variable text is automatically updated. The variable rows is predefined in every combo-box and table. This variable stores the collection of items that can be selected from the combo-box or table. The variable row is also predefined in every combo-box and table where, for each row, it stores the item that can be selected.

Events

Each widget declaration may contain event declarations. Events are triggered when specific actions are executed upon their widgets, and they themselves can execute actions either on data or on other widgets.

The actions executed when an event is triggered are specified using *statements*. A statement is either an action, a conditional statement, an iteration, or a sequence of statements. In GUIML, the conditions in both conditional statements and iterations are specified using OCL expressions, whose context is the underlying data model. Additionally, they can refer to the widget variables. In GUIML, when widget variables are referred within OCL expressions, they are enclosed in dollar signs. Note that each sequence of statements associated to an event is executed as a single *transaction*: either all its statements successfully execute in the given order, or none of them are executed at all.

In concrete syntax, events are declared with the keyword **event**, followed by their types, and the sequence of statements that they execute, enclosed in brackets. In our example we will use two types of events: **onCreate** and **onClick**. The former are triggered when the widgets are created and the latter are triggered when widgets are clicked upon. In particular, a window is created when an open action that has this window as its target is executed. All the other widgets are created immediately after their corresponding containers are created.

Actions

Every event declaration contains a sequence of statements that specifies the actions executed when the event is triggered. These actions can be executed either on objects belonging to the persistence tier or on objects belonging to the presentation tier. The former are called *data actions*, and the latter are called *GUI actions*. Note that some actions may take arguments whose values are only known at run-time, for example a delete action whose argument is the item selected by the user in a combo-box, or an update action whose argument is the number entered by the user in a text field. In GUIML, these values are specified using OCL expressions. Again, the context of these expressions is the underlying data model, but they can also refer to the widget variables.

Next, we briefly describe some of the GUIML data actions and their concrete syntax.

- **Entity create:** It creates a data item in the persistence tier. Its arguments are the *type* of the data item and the *variable* that stores the data item. It is declared by the statement *variable* **:= new** *type*.
- **Entity delete:** It deletes a data item from the persistence tier. Its argument is *object*, which is the data item deleted. It is declared by the statement **delete** [*object*].
- **Attribute read:** It reads the value of a data item's attribute in the persistence tier. Its arguments are the data item *object* whose property is read, the *attribute* read, and the *variable* that stores the value read. It is declared by the statement *variable* **:=** [*object*.*attribute*].

- **Attribute update:** It modifies the value of a data item's attribute in the persistence tier. Its arguments are the data item *object* whose attribute is modified, the *attribute* modified, and the new *value*. It is declared by the statement [*object*.*attribute*] := [*value*].
- **Association-end read:** It reads the collection of items that are linked to an item's association-end in the persistence tier. Its arguments are the data item *object* whose property is read, the association-end *assocEnd* read, and the *variable* that stores the collection read. It is declared by the statement *variable* := [*object*.*assocEnd*].
- **Association-end create:** It creates a link in the persistence tier between two data items. Its arguments are the source data item *srcObject*, the target data item *tgtObject*, and the association-end *assocEnd* through which the target data item is linked to the source data item. An association-end create action is declared by the statement [*srcObject*.*assocEnd*] += [*tgtObject*].
- **Association-end delete:** It deletes a link in the persistence tier between two data items. Its arguments are the source data item *srcObject*, the target data item *tgtObject*, and the association-end *assocEnd* from which the target data item is removed. It is declared by the statement [*srcObject*.*assocEnd*] −= [*tgtObject*].

Finally, we describe some of the GUI actions that are defined in GUIML.

- **Set:** It updates the value of a variable. Its arguments are the name of the *variable* and variable's new *value*. It is declared by the statement *variable* := [*value*].
- **Open:** It opens a window. Its argument is *target*, which names the window opened. Additionally, it may take as arguments any number of pairs ($variable_i$, $value_i$), where $variable_i$ is the name of a variable owned by its *target* window, and $value_i$ is the value that is assigned to $variable_i$ when *target* is opened. It is declared by the statement **open** *target* ($variable_1$: [$value_1$], ... ,$variable_n$: [$value_n$]).
- **Back:** It moves back to the previous window. It is declared using the keyword **back**.
- **Fail:** It forces a rollback in the current transaction, whereby the corresponding statement is not successfully executed. It is declared using the keyword **fail**.
- **Skip:** It does nothing. It is declared using the keyword **skip**.

We now provide an example that illustrates the main elements of the GUIML language: a window ReadPostWI for our chatroom application, where users can read and post messages in a (previously selected) chatroom. As this example will show, GUI models depend on how the application's data is structured — after all, they describe how users interact with this data — but not on the application's access control policy.[1] In our example, this separation of concerns is reflected by the fact that the GUI model for the window ReadPostWI is completely unaware of the security policy for reading and posting messages in our chatroom application.

Example 4 We use GUIML to model the window of our chatroom application where users can read and post messages in a chatroom. This window is named ReadPostWI. It

[1] Of course, in terms of the final application's *usability*, there is a dependency: an application's GUI can end up being unusable precisely because of the application's security policy.

owns a variable chatroomSel that stores a previously selected chatroom (the action that opens the window ReadPostWI will assign a value to this variable). The window ReadPostWI contains four widgets:

- a table ReadPostsTB for visualizing the messages posted in the selected chatroom;
- a text field WritePostEN for writing a new message;
- a button PostBU for posting in the selected chatroom the message written in the text field WritePostEN; and
- a button BackBU for moving back to the previous window.

The model is as follows:

```
1  Window ReadPostWI {
2    //this variable stores the previously selected chatroom
3    Chatroom chatroomSel
4    //this table visualizes the messages posted
5    //in the selected chatroom
6    Table ReadPostsTB {
7      event onCreate {
8          rows := [$ReadPostWI.chatroomSel$.messages] } }
9    //in this text field the user writes her new message
10   TextField WritePostEN {
11     event onCreate { text := [''] } }
12   //by clicking on this button, the user posts her new message
13   //in the selected chatroom
14   Button PostBU {
15     event onCreate { text := ['Post'] } }
16   //by clicking on this button, the user moves back to
17   //the previous window
18   Button BackBU {
19     event onCreate { text := ['Back'] } } }
20 //we continue with this table
21 Table ReadPostWI.ReadPostsTB {
22   //each row of this table shows the body of a message
23   //of the selected chatroom
24   columns {
25     ['Message'] : Label BodyPostLB {
26       event onCreate {
27           text := [$ReadPostWI.ReadPostTB.row$.body] } } } }
28 //we continue with this button
29 Button ReadPostWI.PostBU {
30   event onClick {
31     newPost := new Message
32     newPost.owner += [$ReadPostWI.caller$]
33     newPost.body := [$ReadPostWI.WritePostEN.text$]
34     newPost.chatroom += [$ReadPostWI.chatroomSel$] } }
35 //we continue with this button
36 Button ReadPostWI.BackBU {
37   event onClick { back } }
```

Note that the table ReadPostsTB and the buttons PostBU and BackBU are modeled partially inside the window ReadPostWI and partially outside this window. This is supported by our concrete syntax in order to improve the readability of the GUI models. However, to avoid ambiguities, when a widget is modeled outside its widget container, the widget's global name is used. Note too that the table ReadPostsTB is unaware of the security policy for visualizing messages, which in our running example states that, only registered users are authorized to read messages posted in private chatrooms where they participate. Similarly, the button PostBU is unaware of the security policy for posting messages, which is that only registered users can post messages in public chatrooms and in private chatrooms but, in the latter case, they must also participate in these chatrooms.

2.3 Security-aware GUI Models

2.3.1 Making the security policy explicit

In this section we define a transformation that, for every security model S, produces the security model $S^\flat$, which makes explicit the security policy declared in S. We define this transformation in four steps. Note that, as stated in Remark 1, the following holds at the end of our transformation: for every atomic action *act* and every role r in S, there is exactly one permission in $S^\flat$ (possibly constrained by false) for r to execute *act*.
Step 1: Copy the explicit permissions

- *Atomic actions.* Let *act* be an atomic action. Suppose that there is a permission in S for a role r to execute *act* under a constraint *auth*. Then, there is also a permission in $S^\flat$ for r to execute *act* under the same constraint *auth*.

Step 2: Unfold the security model

- *Action hierarchies.* Let *CA* be a composite action. Suppose that there is a permission in S for a role r to execute *CA* under a constraint *auth*. Then for every atomic action *act* contained in *CA*, there is a new permission in $S^\flat$ for r to execute *act* under the same constraint *auth*.

- *Role hierarchies.* Let *act* be an atomic action and let r and r' be two roles. Suppose that r is a subrole of r' in S, and that there is also a permission in S for r' to execute *act* under the constraint *auth*. There is then a new permission in $S^\flat$ for the role r to execute *act* under the same constraint *auth*.

- *Delete actions.* Let *entity* be an entity. Suppose that there is a permission in S for a role r to delete *entity* under a constraint *auth*. Then for every association-end *assoc* owned by *entity*, there is a new permission in $S^\flat$ for r to execute the action **Delete::***assoc* under the same constraint *auth*.

- *Opposite association-ends.* Let *assoc* and *assoc*$'$ be two opposite association-ends. Let *act* be the action **Create::***assoc*. Suppose that there is a permission in S for a role r to execute *act* under the constraint *auth*. There is then a new permission in $S^\flat$ for the role r to execute **Create::***assoc*$'$ under the constraint that results from simultaneously replacing in *auth* the variable self by target and the variable target by self. Unfolding is similar when *act* is the action **Delete::***assoc*.

Step 3. Add default permissions to the security model

- *Denying by default.* Let r be a role and let *act* be an atomic action. Suppose that there is no permission in $S^\flat$ for the role r to execute *act*. There is then a new permission in $S^\flat$ for the role r to execute *act* under the constraint false. That is, the role r will be denied access to execute *act* in all circumstances.

Step 4. Simplify the resulting security model

- *Disjunction of constraints.* Let r be a role and let *act* be an action. Suppose that there are n permissions in $S^\flat$ for the role r to execute *act*. These n permissions are then simplified to a single permission whose authorization constraint results from disjoining together all the authorization constraints of the n individual permissions.

2.3.2 Making the GUI models security-aware

In this section we describe the heart of our methodology: a model-transformation function *Sec* that, given a GUI model G and a security model S, automatically generates a new GUI model $\mathrm{Sec}(G, S)$. The generated model is identical to G except that it is *security aware* with respect to S. The transformation function *Sec* works by wrapping around every data action *act* in G an if-then-else statement with the following arguments:

- a condition that reflects the constraints associated to the permissions specified in S, for each of the different roles, to execute the action *act*;
- a then-branch that contains the action *act*; and
- an else-branch that contains the action **fail**.

Thus, the semantics of the if-then-else statement ensures that *act* will only be executed if the constraints associated to the corresponding permissions are satisfied. Moreover, this semantics also guarantees that, if these constraints are not satisfied, then the action **fail** will be executed, forcing a rollback in the current transaction.

More specifically, to generate the aforementioned if-then-else statement, the function *Sec* makes use of Remark 1. In particular, for each role r in S, it calls the function $Auth(S, r, act)$ to obtain the expression that ultimately (i.e., when the security policy is made completely explicit) constrains the permission given to r for executing *act*. However, since this expression may contain the variables self, value, target, and caller, the function *Sec* must also replace these variables by the actual arguments of the action *act* (including its actual user). We denote the resulting OCL expression by $\mathrm{Auth}(S, r, act)[args]$, where *args* are the arguments specified in the GUI model for the action *act*. Finally, since different roles may be constrained by different expressions, the condition generated by *Sec* will have the form:

((*r*$_1$ = [*Window*.role] and Auth(*S*, *r*$_1$, *act*)[*args*])
or ... or
(*r*$_n$ = [*Window*.role] and Auth(*S*, *r*$_n$, *act*)[*args*])),

where $r_1, \ldots, r_n$ are all the roles declared in S. (Recall that the actual application's user and its role are always stored in the variables caller and role, which are owned by every window in the GUI model.)

The following examples illustrate the model-transformation function *Sec*. As previously mentioned, the complete, formal account of our methodology, including the model-transformation function *Sec*, is given in [6]. Nevertheless, the interested reader can find in Section 2.3.3 a high-level account of the correctness of *Sec*.

Example 5 Consider line 33 in Example 4. It specifies the third action that will be executed when the button ReadPostWI.PostBU is clicked upon, namely,

```
newPost.body := [ReadPostWI.WritePostEN.text].
```

Recall that := refers to an update action, in this case to the action **Update::**body. The function *Sec* will replace this by the following if-then-else statement:

```
if ((DefaulR = [ReadPostWI.role] and false)
   or
   (UserR = [ReadPostWI.role] and
      ([newPost].owner = [ReadPostWI.caller]
      and [newPost].chatroom.oclIsUndefined())))
then newPost.body := [ReadPostWI.WritePostEN.text]
else fail.
```

To understand the condition generated by *Sec*, note that Auth(ChatRoomSTM, DefaultR, **Update::**body) is equal to false, but that Auth(ChatRoomSTM, UserR, **Update::**body) is equal to self.owner = caller and self.chatroom.oclIsUndefined(). Thus, the function *Sec* must replace the variable self by the newly created message newPost (since this is the object upon which the action **Update::**body will be executed), and the variable caller by ReadPostWI.caller (since this is the user that will execute the action **Update::**body).

Example 6 Consider line 32 in Example 4. It specifies the second action that will be executed when the button ReadPostWI.PostBU is clicked upon, namely,

```
newPost.owner += [ReadPostWI.caller].
```

Recall that += refers to an association-end create action, in this case to the action **Create::**owner. Then, the function *Sec* will replace this line by the following if-then-else statement:

```
if ((DefaulR = [ReadPostWI.role] and false)
    or
    (UserR = [ReadPostWI.role] and
    ([newPost].owner.oclIsUndefined()
      and [ReadPostWI.caller]=[ReadPostWI.caller])))
then newPost.owner += [ReadPostWI.caller]
else fail.
```

To understand the condition generated by *Sec*, note that Auth(ChatRoomSTM, DefaultR, **Create::**owner) is equal to false, but that Auth(ChatRoomSTM, UserR, **Create::**owner) is equal

to self.owner.oclIsUndefined() and target=caller. Thus, the function *Sec* must replace the variable self by the newly created message newPost (since this is the object upon which the action **Create::**owner will be executed), the variable caller by ReadPostWI.caller (since this is the user that will execute the action **Create::**owner), and the variable target also by ReadPostWI.caller (since the actual user is precisely the object that will be added by the **Create::**owner as the owner of the newly created message).

Our next example illustrates how our model transformation *Sec* leads to modularity and separation of concerns whereby the GUI model and the security model can be changed independently, if desired.

Example 7 Suppose that we decide to allow anyone (not only registered users, but also unregistered ones) to post messages in public chatrooms. To update the chatroom application's security-aware GUI model, we just carry out the following steps:

- **Step 1** Change the original chatroom's security model to reflect our security policy changes. We call the new security model PubChatRoomSTM and show below the new permissions for the role DefaultR (i.e., for everybody using the application) to create a message, update the body of a message, and post a message in a chatroom:

```
Role DefaultR {
  Message {
    //everybody can create a new message
    Create
    //everybody can change the body of
    //any unowned message
    //provided it is not yet posted anywhere
    if self.owner.oclIsUndefined()
      and self.chatroom.oclIsUndefined()
      then Update::body
    //everybody can post in a public chatroom
    //any unowned message
    //provided it is not yet posted anywhere
    if self.owner.oclIsUndefined() and target.public
      and self.chatroom.oclIsUndefined()
      then Create::chatroom } }
```

- **Step 2** Apply our model transformation to the original chatroom GUI model and the modified chatroom security model to generate the updated security-aware GUI model. We show below the result of this transformation for line 33 in Example 4.

```
if ((DefaultR = [ReadPostWI.role] and
     ([newPost].owner.oclIsUndefined()
       and [newPost].chatroom.oclIsUndefined()))
    or
    (UserR = [ReadPostWI.role] and
     (([newPost].owner.oclIsUndefined()
       and [newPost].chatroom.oclIsUndefined())
     or
```

```
      ([newPost].owner = [ReadPostWI.caller]
        and [newPost].chatroom.oclIsUndefined()))))
then newPost.body := [ReadPostWI.WritePostEN.text]
else fail.
```

It is interesting to compare this result with the one explained in Example 5 for the case of the security model ChatRoomSTM. To understand the differences, note that Auth(ChatRoomSTM, DefaultR, **Update::**body) is equal to false, but that Auth(PubChatRoomSTM, DefaultR, **Update::**body) is equal to self.owner.oclIsUndefined() and self.chatroom.oclIsUndefined(). Also, recall that UserR inherits all permissions from DefaultR and, in particular, its new permission for updating the body of a message, which is constrained by Auth(PubChatRoomSTM, DefaultR, **Update::**body).

Note that in this example, the function *Sec* may generate conditions that can be further simplified. However, for the sake of illustration, here and elsewhere, we show the results of *Sec* without further simplification.

2.3.3 Correctness of our Model Transformation

We sketch here the correctness of our model transformation *Sec*, which is defined relative to the semantics of GUI models. Full details are provided in [6]. We define the semantics of GUI models by first giving a set of inference rules that defines a transition relation $\longrightarrow$ between triples of the form $\langle stm, I, \theta \rangle$, where *stm* is a statement, I is a scenario (i.e., an instance of the underlying data model), and θ represents a state of the widget variables. We provide inference rules for each possible statement: namely, for every type of data action and GUI action (base cases), and for arbitrary sequences of statements, conditional statements, and iterator-statements (inductive cases). In particular, for data actions, the inference rule has the form

$$\frac{}{\langle act(args), I, \theta \rangle \longrightarrow \langle \textbf{skip}, \mathrm{res}(I), \mathrm{res}(\theta) \rangle} \,,$$

where:

- *args* are the arguments of the data action *act*,
- $\mathrm{res}(I)$ specifies the scenario that results from executing $act(args)$ in the scenario I and widget variable state θ, and
- $\mathrm{res}(\theta)$ specifies the widget variables' new state after executing $act(args)$ in the scenario I and widget variable state θ.

Crucially, no inference rule leading to **skip** is defined for the GUI action **fail**.

We then define the *operational semantics* of an event *ev* that executes the actions specified by an statement *stm* as the set of all the transitions

$$\langle stm, I, \theta \rangle \longrightarrow^* \langle \textbf{skip}, I', \theta' \rangle \,,$$

where $\longrightarrow^*$ is the reflexive-transitive closure of $\longrightarrow$.

By definition, this operational semantics for events is *security-unaware*: it does not respect the authorization constraints that, according to the given security model, should constrain the execution of data actions. To provide a *security-aware* operational semantics for events, we define the *security-aware versions* of the inference rules. In particular, given

a security model S, for each role r, and for each type of data action act, the security-aware version of the inference rule for act and r has the form

$$\frac{[\![(\mathrm{Auth}(S, r, act)[args])\theta]\!]^I = \mathsf{true}}{\langle act(args), I, \theta\rangle \longrightarrow \langle \mathbf{skip}, \mathrm{res}(I), \mathrm{res}(\theta)\rangle}\,,$$

where:

- $[\![expr]\!]^I$ denotes the value of the expression $expr$ in the scenario I; and, therefore,
- $[\![(\mathrm{Auth}(S, r, act)[args])\theta]\!]^I$ denotes the evaluation in the scenario I of the authorization $\mathrm{Auth}(S, r, act)$ that constrains the only permission that, according to Remark 1, ultimately allows users with the role r to execute the action act, given that arg are the arguments of act and θ is the state of the widget variables.

These security-aware inference rules define the transition relation $\longrightarrow_S$. Finally, given a security model S, we define the *security-aware operational semantics* of an event ev that executes the actions specified by an statement stm as the set of all the transitions

$$\langle stm, I, \theta\rangle \longrightarrow_S^* \langle \mathbf{skip}, I', \theta'\rangle\,.$$

The theorem below formalizes the *correctness* of our model-transformation function *Sec*. It states that evaluating a statement transformed by *Sec* following the security-unaware operational semantics, returns the same result as evaluating the original statement using the *security-aware* semantics. Hence the transformed statement respects the authorization constraints formalized in the underlying security model.

Theorem Let S be a security model and let stm be a statement. Then, for every scenario I, and every widget variables' state θ,

$$\langle \mathrm{Sec}(stm, S), I, \theta\rangle \longrightarrow^* \langle \mathbf{skip}, I', \theta'\rangle \Longleftrightarrow \langle stm, I, \theta\rangle \longrightarrow_S^* \langle \mathbf{skip}, I', \theta'\rangle.$$

Chapter 3

Supporting model-driven development

Security-aware GUI models are platform independent and can be mapped to implementations employing different technologies. This includes desktop applications, web applications, and mobile applications. As part of our work, we built the ActionGUI toolkit [1], which automatically generates web-based data-management applications from security-aware GUI models.

After a brief report on the current status of the ActionGUI toolkit, we report in this Chapter on our experience developing the Eye OCL Software (EOS) evaluator, a Java component for efficient OCL evaluation on medium-large scenarios, and we also explore various approaches for evaluating OCL expressions on really large scenarios. Then, we propose a mapping from a subset of OCL into first-order logic (FOL) and use this mapping for checking the unsatisfiability of sets of OCL constraints. We argue that our mapping is both simple, since the resulting FOL sentences closely mirror the original OCL constraints, and practical, since we can use automated reasoning tools, such as automated theorem provers and SMT solvers, to automatically check the unsatisfiability of non-trivial sets of OCL constraints. Finally, we propose a methodology for reasoning about fine-grained access control policies, whose authorization constraints are specified in OCL, using the aforementioned mapping from OCL to FOL.

3.1 The ActionGUI toolkit

The ActionGUI toolkit features model editors for constructing and manipulating data, security, and GUI models. These editors share our own OCL parser, which takes as additional input the variables introduced by the different models, along with their respective types: in the case of security models, the variables self, caller, target, and value, and in the case of GUI models, all the given widget variables. Crucially, the ActionGUI toolkit implements our model transformation to generate security-aware GUI models. Finally, it includes a code generator that, given a security-aware GUI model, produces a web application based on the following, standard three-tier architecture.

1. Presentation tier (also known as front-end): Users access web applications through standard web browsers, which render the content (HTML and JavaScript) dynamically provided by the web server.

2. Application tier: The toolkit generates Java Web Applications, implemented using the Vaadin framework [82]. The applications run in a web server (such as Tomcat or

GlassFish), process client requests and, generate content, which is sent back to the client for rendering. They may also manipulate data stored in the persistence tier. When processing client requests, the generated application *interprets* its underlying security-aware GUI model. In particular, it performs the required security checks before modifying any data stored in the persistence tier or sending any data to the presentation tier. This involves, of course, dynamically evaluating the OCL expressions appearing in the security-aware GUI model.

3. Persistence tier (also known as data tier or back-end): The generated application manages information stored in a database. For each application, the toolkit generates the corresponding database schema from the application's data model. This schema contains the SQL commands to create the database.

As a model-driven development tool, the ActionGUI toolkit produces its best results when the intended data-management application's functionality can be reduced to CRUD actions and its dynamics consists of navigating through windows and exchanging information with the underlying database. For applications in this category, ActionGUI automatically generates the complete implementation from the corresponding data, security, and GUI model. Note that calling CRUD actions is modeled in GUIML using data actions, and navigating and passing information through windows is modeled using GUI actions, namely, **open**, **back**, and **set**. Of course, some data-management applications will require additional functionality, for example, the possibility of sending emails, printing tables, or exporting data in specific formats. As expected, the ActionGUI toolkit does not generate code for these non-CRUD methods. Instead, it includes their implementation — which must be provided by the application developer — in the generated application and when the application needs to interpret one of these methods, it simply calls the method provided.

3.2 Evaluating OCL expressions

In [33] we report on our experience developing the Eye OCL Software (EOS) evaluator, a Java component for efficient OCL evaluation. We first motivate the need for an efficient implementation of OCL in order to cope with novel usages of the language. We then discuss some aspects that, based on our experience, should be taken into account when building an OCL evaluator for medium-large scenarios. Finally, we explore various approaches for evaluating OCL expressions on really large scenarios.

3.2.1 Motivation

As part of our research, we have looked at different usages of OCL beyond its "initial requirements as a precise modeling language complementing UML specifications." Two related applications have drawn our interest [16, 3, 4, 35, 8], both having to do with using OCL to analyze user-defined models by evaluating queries on the corresponding instances of their metamodels. Since these instances typically contain a large number of elements, evaluating expressions on them comes at a high computational cost.

Consider, for example, the use of OCL to express metrics for Java programs (this application was suggested to us by members of the Triskell group at IRISA, France). The scenarios on which the program metrics will be evaluated, are the instances of the Java metamodel corresponding to the programs: thus, the larger the programs the larger the

scenarios[1] and, consequently, the higher the computational cost of evaluating the program metrics.

We report in [33] on our experience developing the Eye OCL Software [41] (EOS) component, an OCL evaluator designed with the goal of performing efficient evaluation of OCL expressions on medium-large size scenarios. In particular, we discuss i) the need for an efficient implementation on OCL in order to cope with the novel usages of the language; ii) the aspects that we have taken into consideration to improve the efficiency of the EOS evaluator on medium-large scenarios; iii) the limits of the current OCL implementations for dealing with really large scenarios. Although we include the results of applying a benchmark to several OCL evaluators, this Section is not a comparative study (see [54] for a study of this kind). In fact, the results are included here only to show the performance of some OCL tools on medium-large scenarios and to illustrate the aspects that we consider that should be taken into account when implementing an efficient OCL evaluator for medium-large scenarios. Interestingly, this quality —OCL engine efficiency on medium-large scenarios— is not covered by the benchmark proposed in [54]: in fact, the largest scenario proposed for testing OCL engine efficiency in [55] contains only 42 objects and 42 links among them. Furthermore, despite the results of our benchmark, this Section is not a promotional brochure for our EOS component: as a "product", our OCL evaluator is still in its infancy. To motivate the need for OCL engines that can efficiently evaluate expressions on medium-large size scenarios, we show in Table 3.1 the time that currently takes to evaluate two given OCL expressions on three different, small-medium size scenarios for a number of OCL evaluators: namely, USE 2.4.0 [40], RoclET [2], OCLE [30], MDT OCL [59], and EOS [41].[2] The tests were run on a laptop computer, with Windows XP Professional installed, a processor Intel Pentium M 2.00GHz 600MHz, and 1GB of RAM. Also, in the case of the EOS and USE evaluators, we run the JVM with its parameters `-Xms` and `-Xmx` set to 1024m.

The scenarios considered in these tests are instances of the model Library shown in Figure 3.1: each scenario is referenced by a number n, which also indicates its "size"; more precisely, for each n, the scenario $\#n$ is a library that contains exactly 10^n books, each book with a unique title different from "Hobbit". The OCL expressions used in these tests are

(3.1) `Book.allInstances()->forAll(b|b.title <> 'Hobbit')`

(3.2) `Book.allInstances()->forAll(b1,b2|b1 <> b2 implies b1.title <> b2.title).`

The first expression says that the library does not contain any book titled "Hobbit", while the second one says that the library does not contain two different books with the same title. Obviously, the cost of evaluating these expressions depends on the number of books in the library and the cost of accessing, and storing for later use, information about these books. For example, in order to evaluate the expressions (3.1) and (3.2) on scenario #3 we have to perform, respectively, 10^3 and $2 \times 10^3 \times 10^3$ times the operations of accessing and storing a book's title. But this is precisely one of the challenges for OCL engines when evaluating expressions on medium-large size scenarios: namely, to efficiently access the information contained in, possibly, all the objects that populate the scenarios.

[1] As an example, the SpoonEMF application, developed by the Triskell group generates, for a standard Java program with 10 lines, a scenario with 113 objects; for a program with 100 lines, one with 1180 objects; and for a program with 500 lines, one with 3470 objects.

[2] In the case of USE, we have run our experiments using its version 2.4.0. For MDT OCL, we have run the experiments using the plug-in "OCL Interpreter", version 1.2.0v200805130238. Finally, for our experiments in OCLE, we have used its version 2.0.

Scenario	*Expression*	RoclET	OCLE	MDT OCL	EOS	USE
#2	(1)	< 1s	< 1s	< 1s	0ms	230ms
	(2)	> 10m	≈ 4s	< 1s	50ms	240ms
#3	(1)	–	≈ 3s	< 1s	10ms	240ms
	(2)	–	> 10m	≈ 4s	841ms	1s342ms
#4	(1)	–	–	< 1s	20ms	261ms
	(2)	–	–	$\approx$ 5m12s	1m18s	2m12s

Table 3.1: Evaluating performance on medium-large scenarios.

Figure 3.1: The model Library.

Notice that in Table 3.1 we use $> t$ to indicate that we stopped an experiment after having passed time t without obtaining the result. Also, we use a dash to indicate that we did not run the experiment, because we already stopped the execution of a "simpler" one. Finally, since RoclET, OCLE and MDT OCL do not report their execution time in milliseconds, we use < 1s to indicate that the result of an evaluation was output in less than 1 second, and we use $\approx t$ to indicate that the result of an evaluation was output approximately in time t.

3.2.2 Measuring the cost of evaluating expressions

Although the computational cost of evaluating a particular OCL expression on a given scenario, obviously depends on the algorithms and data structures used to implement each tool, based on our experience, there are two measurements worthwhile considering before launching the evaluation process: first, the maximum number of times that objects' properties will be accessed and, second, the maximum size of the collections that will be built. In the case of medium-large size scenarios, the challenge for OCL engines is that these measurements typically return large numbers.

To illustrate this challenge, we show in Table 3.2 the performance of MDT OCL, EOS, and USE when evaluating different iterator-expressions whose evaluation require accessing many times objects' properties and/or building large collections. All the expressions in Table 3.2 were evaluated on the same scenario, namely, an instance MyLibrary of the model Library shown in Figure 3.1 which contains 10^3 authors, each author with 10 different books, and each book with a title different from "Hobbit".[3] For the sake of the experiment, we artificially increased the size of the collections to be iterated upon: in particular, in Table 3.2, the terms p_1, p_2, and p_3 refer, respectively, to the expressions "Book.allInstances().author.books", "p_1.author.books", and "p_2.author.books",

which on the scenario MyLibrary evaluate to collections with 10^5, 10^6, and 10^7 books, respectively.[4] The iterator-expressions in Table 3.2 were evaluated on a laptop computer,

[3] In the case of MDT OCL, the scenario MyLibrary was loaded from an XMI file; for EOS, it was built using a Java program that simply calls the appropriate EOS's interface methods for defining the scenario; and for USE, it was loaded from a file that contained the appropriate USE commands to build the scenario.

[4] A more "natural" approach will be to consider scenarios with larger number of books but, as we will

with Windows XP Professional installed, a processor Intel Pentium M 2.00GHz 600MHz, and 1GB of RAM. Also, in the case of the EOS and USE evaluators, we run the JVM with its parameters `-Xms` and `-Xmx` set to 1024m. For the reason explained above, in the case of the MDT OCL evaluator, all times shown in Table 3.2 are approximated ($\approx$).[5] The *Error* evaluation time indicates that we could not evaluate the expression due to an OutOfMemoryError in Java.

Let $exp(p_i)$, with $i \in \{1, 2, 3\}$, be the time that takes to evaluate an iterator-expression exp on the collection generated by evaluating the term p_i. With respect to the performance of MDT OCL, EOS, and USE, Table 3.2 shows that, for the same exp, the time $exp(p_{i+1})$ is approximately $10 \times exp(p_i)$ in these tools. Table 3.2 also shows that, for the same p_i, the time $exp(p_i)$ depends for these tools on the number of accesses to objects' properties that are required to evaluate the body of the iterator-expression exp; we have organized accordingly the expressions in three groups: A, B, and C. Consider, for example, the evaluation times for the first expression in each group. Finally, Table 3.2 shows that, again for the same p_i, the time $exp(p_i)$ depends as well for these tools on the size of the collection that need to be built. Consider, for example, the evaluation time for the first two expressions in Group C: notice that to evaluate

$$p_3 \text{->collect(x|x.author.books.title)->size()} \tag{3.3}$$

will require to allocate memory for storing a collection with 10^8 titles, while evaluating

$$p_3 \text{->collect(x|x.author.books.title->size())->sum()} \tag{3.4}$$

will *only* require to allocate memory for storing a collection with 10^7 integers. Using these and similar expressions, we regularly use the above mentioned measurements, namely, the maximum number of times that objects' properties will be accessed and the maximum size of the collections that will be built, to check the performance of the EOS evaluator and look for possible optimizations.

3.2.3 The implementation of the EOS evaluator

ITP/OCL [31] is a rewriting-based OCL evaluator, based on an executable equational semantics for OCL [49]. Although this tool performs reasonably well on small-medium size scenarios, its performance does not scale up to medium-large size scenarios. Prompted by our interest on applications that require efficient OCL evaluation on medium-large size scenarios, we decided to implement the Eye OCL Software (EOS) evaluator, a Java component whose design follows the key ideas behind the ITP/OCL tool. Its implementation includes an OCL parser (which uses SableCC [53]) and an OCL evaluator, the latter consisting of about 7K lines of Java code. It covers the full OCL 2.0 Standard Library [72], with the exception of the OclMessage type and its operations. With the idea of making it as 'pluggable' as possible, the EOS component is not based on any particular (meta)modeling framework: its public interface provides methods to insert elements, one-by-one, into user-models and scenarios, and to input the expressions to be evaluated as

discuss in Section 3.2.4, none of the tools support well the loading of scenarios with more than 10^6 objects.

[5] We have also run the same experiments using a desktop computer, with Window Vista Business installed, a processor Intel Core 2 Quad CPU Q9300 2.50GHz 2.50 GHz, and 2GB of RAM. Again, in the case of the EOS and USE evaluators, we run the JVM with its parameters `-Xms` and `-Xmx` set to 1024m. In the case of EOS and MDT OCL the results were similar to those shown in Table 3.2. In the case of USE, however, the evaluations were completed, approximately, in half the time taken by the single-core test machine.

		MDT	EOS	USE
p_1	->size()	< 1s	30ms	1s12ms
p_2		< 1s	190ms	7s330ms
p_3		≈ 5s	931ms	1m22s
Group A		MDT	EOS	USE
p_1	->collect(x|x.title)->size()	< 1s	80ms	1s212ms
p_2		≈ 1s	391ms	10s84ms
p_3		≈ 18s	3s 896ms	1m48s
p_1	->collect(x|x.title <> 'Hobbit')->size()	< 1s	90ms	981ms
p_2		≈ 2s	481ms	8s432ms
p_3		≈ 20s	4s 516ms	1m30s
Group B		MDT	EOS	USE
p_1	->collect(x|x.author.books)->size()	< 1s	240ms	6s810ms
p_2		≈ 6s	2s 140ms	1m42s
p_3		≈ 58s	17s 736ms	*Error*
p_1	->collect(x|x.author.books->includes(x))->size()	< 1s	221ms	3s565ms
p_2		≈ 7s	1s 893ms	32s677ms
p_3		≈ 1m 1s	17s 906ms	5m32s
p_1	->forAll(x|x.author.books->includes(x))	< 1s	251ms	3s475ms
p_2		≈ 5s	1s 963ms	32s6ms
p_3		≈ 53s	17s 30ms	5m25s
p_1	->select(x|x.author.books->includes(x))->size()	< 1s	260ms	3s685ms
p_2		≈ 5s	2s 13ms	35s411ms
p_3		≈ 55s	17s 605ms	5m51s
Group C		MDT	EOS	USE
p_1	->collect(x|x.author.books.title)->size()	≈ 2s	290ms	8s412ms
p_2		≈ 20s	2s 573ms	1m27s
p_3		≈ 3m 17s	23s 684ms	*Error*
p_1	->collect(x|x.author.books.title->size())->sum()	≈ 2s	270ms	4s957ms
p_2		≈ 19s	2s 274ms	48s660ms
p_3		≈ 3m 15s	20s 840ms	10m54s
p_1	->forAll(x|x.author.books.title->excludes('Hobbit'))	≈ 2s	280ms	4s777ms
p_2		≈ 20s	2s 604ms	46s286ms
p_3		≈ 3m 15s	22s 802ms	7m52s

Table 3.2: Evaluating performance: MDT OCL, EOS, and USE.

	MDT OCL			EOS		USE	
Scenario	XMI	Mem	Time	Mem	Time	Mem	Time
#3	50KB	111MB	< 1s	20MB	40ms	88MB	1s
#4	500KB	112MB	< 1s	22MB	661ms	116MB	35s
#5	5MB	125MB	≈ 45s	50MB	2m36s	209MB	8m25s
#6	50MB		> 20m		> 20m		> 20m

Table 3.3: Evaluating loading cost: MDT OCL, EOS, and USE.

strings of ASCII characters. This decision allowed us also to design the EOS's data structure for internally storing user-models and scenarios in such a way that objects' properties are efficiently accessed. The other possible novelty in its implementation is that, before evaluating a collect expression, we try to (over)estimate the size of the resulting collection and allocate memory in advance. The rest of the EOS implementation is rather straightforward: OCL iterator-expressions are executed using Java for/while loops and standard OCL operations, when possible, are executed using the appropriate Java operators. As expected, expressions are evaluated in EOS following an eager strategy: in particular, collection-expressions are fully evaluated and their resulting elements are all allocated in memory.

3.2.4 Limitations

To evaluate expressions on really large scenarios, we need first to solve the problem of loading the scenarios in the OCL evaluators. To illustrate this challenge, we show in Table 3.3 the time and memory taken by MDT OCL, EOS, and USE, when loading different scenarios of the model Library. Each of the scenarios is identified by a number n, which also indicates its "size": more precisely, for each n, the scenario $\#n$ exactly contains 10^n books. Notice that for scenario #6, with 10^6 books, none of the tools were able to finish in less than 20 minutes.[6]

So far, we have explored two different approaches for addressing this problem, both based on the representation of user-models and scenarios as relational databases. The first approach consists on modifying the EOS evaluator so as to look for the information contained in the scenarios directly in its database representation. The advantage of this approach is that it only requires modifying the evaluation of dot-expressions in the expected way: namely, accessing the value of an object's attribute or the value of an object's association-end will be now implemented as a basic SQL `select`-query. The concrete form of these queries depends, of course, on the mapping used to represent models as relational databases (see, for example, [75, 80] and [56]). To check the feasibility of this approach, we modified the EOS evaluator accordingly: unfortunately, the cost of evaluating dot-expressions through the JDBC driver was so high that it made impractical the use of the modified EOS evaluator for evaluating expressions on medium-large scenarios.[7]

The second approach consists on translating OCL expressions into expressions in a query language already available for relational databases. To the best of our knowledge, the most interesting results in this line are discussed in [47, 58] and provide the foundations of the OCL2SQL tool [57, 81]. However, the solution offered in [47, 58] is not satisfactory yet. First, it only considers a restricted subset of the OCL language: in particular, it

[6] With respect to the "extra" time taken by the EOS tool to store scenarios, compared to MDT OCL, it is possibly due to the extra computation required to store scenarios in the EOS internal data structure.

[7] For this experiment, we mapped each class to a table whose columns correspond to its attributes, and each association to a table whose columns correspond to its association-ends.

Scenario	Time
# 1	≈ 0m25s
# 2	≈ 45m

Table 3.4: Evaluating performance: OCL2SQL.

cannot deal with iterators, tuples or nested collections. Second, it only applies to Boolean expressions and not to arbitrary queries. Finally, the "complexity" of the SQL expressions resulting from this translation is so high that makes also impractical its use for evaluating expressions on medium-large scenarios.[8]

3.3 Checking the unsatisfiability of OCL expressions

In [34] we propose a mapping from a subset of OCL into first-order logic (FOL) and use this mapping for checking the unsatisfiability of sets of OCL constraints. We argue in [34] that our mapping is both simple, since the resulting FOL sentences closely mirror the original OCL constraints, and practical, since we can use automated reasoning tools, such as automated theorem provers and Satisfiability Modulo Theories (SMT) solvers to automatically check the unsatisfiability of non-trivial sets of OCL constraints. SMT generalizes boolean satisfiability (SAT) by incorporating equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other useful first-order theories [5].

3.3.1 Motivation

The lack of tool support for OCL was pointed out in [29] as a main cause for the limited adoption of the language in industry. Since then, many initiatives have been brought to fruition and their outcomes are available to designers (see [46]). Among the tool categories that have received significant attention are:

- *Parsers*: to check the syntactical well-formedness of an expression: e.g., the Dresden OCL 2.0 parser [62, 81].
- *Evaluators*: to obtain the value of an expression within a contextual model: e.g., USE [40], MDT OCL [59], and EOS [33].
- *Translators*: to map (for different purposes) an expression into a (logically equivalent) expression in other languages and/or formalisms: e.g.,
  - OCL2SQL [57, 81] maps OCL constraints into SQL queries;
  - UMLtoCSP [23, 22] maps OCL constraints into constraint programming expressions, to support automated bounded verification of UML class diagrams, annotated with OCL constraints;
  - MOMENT [27] and ITP-OCL [32] map OCL into equational logic (although using different approaches) to support automated evaluation of OCL expressions using term-rewriting.
  - KeY [15] includes a mapping of OCL into first-order logic to allow interactive reasoning about UML diagrams with OCL constraints.

[8] The OCL2SQL's main developer has confirmed that the efficiency of the OCL2SQL tool has not been tested on medium-large scenarios (e-mail communication, May 2008).

– HOL-OCL [17, 18] maps OCL into higher-order logic also to allow interactive reasoning about UML diagrams with OCL constraints.

The work presented in [34] belongs to the third category: it proposes a mapping from OCL to first-order logic, which is defined with the purpose of supporting (*unbounded*) unsatisfiability checks for OCL expressions using *automated* reasoning tools. In our view, being able to check the *unsatisfiability* of (sets of) OCL expressions is a powerful tool, since it will allow modelers to (among other tasks):

- Verify class invariants, by checking that they logically imply the expected constraints/properties;
- Verify method preconditions, by checking that the class invariants do not logically imply their negations; and
- Verify method postconditions, by checking that they do not logically imply the negation of (any of) the class invariants.

However, also in our view, what will make an unsatisfiability checker not only powerful, but also practical, is being *automated*. Given the undecidable nature of the full OCL language, one can only expect to have an automated unsatisfiability checker for a large class of OCL expressions. In [34] we do not attempt to define how large and/or interesting is the class of unsatisfiable OCL expressions, that we are able to check automatically. Nevertheless, we argue that our mapping is both simple, since the resulting FOL sentences closely mirror the original OCL constraints, and practical, since we can use existing automated theorem provers (e.g., Prover9 [67]) and/or SMT solvers (e.g., Yices [48]) to automatically check the unsatisfiability of non-trivial sets of OCL constraints.

3.3.2 Unsatisfiability of OCL constraints

The notion of *unsatisfiability* that we propose emphasizes the logical meaning of OCL constraints; in fact, it basically translates to OCL the standard notion of unsatisfiability for logic formulas. There are other notions of satisfiability/unsatisfiability for OCL constraints in the literature, which we will briefly discuss at the end of this section.

In what follows, we denote by OCL *constraint* any OCL expression of type Boolean. We do not assume that instances of models always have a finite a number of elements.

Definition 1 *Given a model (class diagram) $\mathcal{M}$, and a set of OCL constraints Φ, we say that Φ is* $\mathcal{M}$-unsatisfiable *if and only if there does not exist an $\mathcal{M}$-instance (object diagram) $\mathcal{O}$ on which every constraint in Φ evaluates to* true.

To illustrate this notion, we introduce the following example. Table 3.5 shows a list of OCL constraints: they all refer to the simple model Library2 shown in Figure 3.2. In this model, libraries contains *Books* and books have *Authors*, *pages*, and an *ISBN* code.

According to Definition 1, the following subsets (among others) of the constraints shown in Table 3.5 are *Library2*-unsatisfiable: {1,2}, {1,8}, {1,10}, {2,3}, {2,4}, {2,5}, {2,6}, {7,8}, {11, 13}, {12}, {14}, and {15}.

Notice that the subset {9,10,11} is not *Library2*-unsatisfiable: a library with just one book will satisfy these constraints. On the other hand, the subset {9,10,11,16} is indeed *Library2*-unsatisfiable. Notice also that the subset {17} is not *Library2*-unsatisfiable: a library with an infinite number of books will satisfy this constraint. On the other hand, the subset {17, 18} is indeed *Library2*-unsatisfiable.

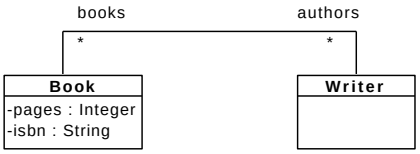


Figure 3.2: The model Library2.

1. Book.allInstances()−>isEmpty().
2. Book.allInstances()−>exists(x|x.pages > 300).
3. Book.allInstances()−>forAll(x|x.pages < 300).
4. Book.allInstances()−>select(x|x.pages > 300)−>isEmpty().
5. Book.allInstances()−>reject(x|x.pages <= 300)−>isEmpty().
6. Book.allInstances()−>collect(x|x.pages)−>asSet()−>forAll(i|i < 300).
7. Book.allInstances()−>forAll(x|x.authors−>isEmpty()).
8. Author.allInstances()−>exists(a|a.books−>notEmpty()).
9. Book.allInstances()−>forAll(x,y|x<>y implies x.isbn<>y.isbn)).
10. Book.allInstances()−>exists(x|Book.allInstances()−>excluding(x) −>forAll(y|y.isbn=x.isbn)).
11. Book.allInstances()−>notEmpty().
12. Book.allInstances()−>exists(x|Book.allInstances()−>excludes(x)).
13. Book.allInstances()−>forAll(x|Book.allInstances()−>excluding(x)−>includes(x)).
14. Book.allInstances()−>exists(x|Book.allInstances()−>excluding(x)−>includes(x)).
15. Book.allInstances()−>collect(x|x.authors)−>asSet()−>exists(y|y.books −>isEmpty())).
16. Book.allInstances()−>size() > 1.
17. Book.allInstances()−>forAll(b|Book.allInstances()−>exists(x|x.pages > b.pages)).
18. Book.allInstances()−>size() = 2.

Table 3.5: List of constraints.

As mentioned before, other notions of satisfiability/unsatisfiability of UML models with OCL constraints can be found in the literature. In particular, the notions used in [23, 24] are those of *weak* and *strong satisfiability* (and related notions are also introduced in [18]). Weak satisfiability means that there exists a finite instance of the model in which at least one class is populated with at least one element. Strong satisfiability means that there exists a finite instance of the model in which all its classes are populated with at least one element. Notice that, if a set of constraints is unsatisfiable (in our sense), then it can not

be weak nor strong satisfiable. On the other hand, if a set of constraints is not weak nor strong satisfiable, it does not imply that is unsatisfiable (in our sense).

3.3.3 A mapping from OCL to FOL

In this section we provide a high-level description of a mapping from a subset of OCL into first-order logic (FOL). The interested reader can find in [34] the full details of this definition. Given a set of OCL constraints, our mapping generates a set of FOL formulas such that, if the resulting set is unsatisfiable, then the original set is also unsatisfiable.

OCL constraints specify properties that must be satisfied by a model. In order to do so in a concise way, OCL provides different constructors to refer to specific collections of elements. In a nutshell, our mapping is defined recursively over the structure of OCL expressions:

- Boolean-expressions are translated to formulas, which essentially mirror their logical structure; Integer-expressions are basically copied; at this point, we do not consider String-expressions.
- Collection-expressions are translated to predicates, whose meaning is defined by additional formulas generated by the mapping; at this point, we only consider Set-expressions.
- Association-ends are translated to predicates, which are also defined by formulas generated by the mapping; at this point, we do not consider qualified associations.
- Attributes are translated to functions, which are left undefined by the mapping.

Checking unsatisfiability

A crucial advantage of our mapping is that the resulting formulas can be checked for unsatisfiability using *automated* reasoning tools, such as automated theorem provers and SMT solvers. To illustrate our point, we have tried to automatically prove the unsatisfiability of different subsets of the constraints shown in Table 3.5 using Prover9 [67] (an automated theorem prover) and Yices [48] (an SMT solver). The results are shown in Table 3.6. The symbol $\surd$ indicates that the corresponding tool was able to prove the unsatisfiability of the input set of formulas, while the symbol $\bullet$ indicates that the tool concluded without finding a proof. Both tools finished each task in less than a second, running on a standard laptop computer. In our experiments, we used both provers with their default (command) options.

Prover9 [67] is a resolution/paramodulation automated theorem prover for first-order and equational logic. It uses two default limits which, although good in practice, can prevent proofs from being found. Not surprisingly (since it does not support integer arithmetic), Prover9 could not automatically prove the unsatisfiability of some of the subsets of constraints in our experiment.

Yices [48] is a high-performance SMT solver that decides the satisfiability of propositional formulas that mix uninterpreted function symbols and equality with interpreted symbols from various theories, in particular for linear real and integer arithmetic, but also for recursive data types, tuples, records, lambda expressions and quantifiers among others. Of course, when dealing with quantifiers, SMT solvers cannot be complete, and may return *unknown* after a while, meaning that they can neither prove the quantified formula to be unsatisfiable, nor can they find an interpretation that makes it satisfiable. However, Yices was able to automatically prove the unsatisfiability of all the subsets of constraints

	{1,2}	{1,8}	{1,10}	{2,3}	{2,4}	{2,5}	{2,6}	{7,8}	{11,13}	{12}	{14}	{15}
Prover9	√	√	√	•	√	•	•	√	√	√	√	√
Yices	√	√	√	√	√	√	√	√	√	√	√	√

Table 3.6: Case study: checking unsatisfiability.

in our experiment. This result is certainly encouraging for our purposes; moreover, we expect to take advantage of its decision procedures for recursive data types, tuples, and records, in order to prove the unsatisfiability of more complex subsets of OCL constraints.

3.3.4 Limitations

The mapping from OCL to first-order logic proposed here, supports the direct use of SMT solvers to check the satisfiability of OCL expressions, but under specific restrictions both on the expressions that are allowed, and on the instances of data models that are checked. Not surprisingly, some of these restrictions are imposed to avoid the problem of coping with undefinedness in OCL. In particular, our mapping i) only considers instances where all attributes are defined and ii) does not allow expressions containing operators that can generate undefined values. In [39] modifications of the previous mapping are proposed that, although grounded on the same principles underlying our mapping, are designed to overcome the limitations of the latter with regard to undefinedness in OCL. Let us explain the key difference in a nutshell. Under restrictions i) and ii) above, a Boolean OCL expression can only evaluate to either true or false. Thus, to reason in this context using Boolean OCL expressions it is sufficient to define a mapping that formalizes when a Boolean OCL expression evaluates to true. This is precisely what we do in the mapping propose here. However, in the presence of undefinedness, Boolean OCL expressions can also evaluate to null or invalid. To cope with this fact, [39] defines four different mappings which formalize, respectively, when a Boolean OCL expression evaluates to true, when to false, when to null, and when to invalid.

In the near future, we plan to further extend the mapping from OCL to first-order logic [34, 39] to deal with larger subsets of OCL. First, we should be able to deal with generalizations. The idea here is to add, by default, the expected sentences formalizing that every element in the sub-class collection also belongs to the super-class collection. Second, we should be able to deal with bags. We have not yet found a solution for translating Bag-expressions using predicates, as we do for Set-expressions (the obvious one of using an additional argument, indicating the number of occurrences of a given element in the bag, does not seem to work). Third, we should be able to deal with size-expressions. We do not have yet a general solution for this: dealing with constraints like (16) and (18) in Table 3.5 and, in general, with constraints that simply restrict the multiplicity of collections, is rather simple; defining a mapping for arbitrary size-expressions appearing anywhere inside constraints, requires further investigation. Finally, it would be interesting to define a mapping for general Collection-expressions and for Tuple-expressions, and to explore the capabilities of SMT solvers to automatically check the unsatisfiability of the resulting formulas. In the long term, we should prove, of course, the correctness of our mapping with respect to a formal semantics of the OCL language.

3.4 Reasoning about access control policies

In this Section we present a novel, tool-supported methodology for reasoning about fine-grained access control policies (FGAC). We also briefly report on our experience using

the Z3 SMT solver [45] for automatically proving non-trivial properties about FGAC policies.

The key component of our methodology is the mapping [34, 39] from OCL to first-order logic discussed in Section 3.3.3, which allows one to transform questions about FGAC policies into satisfiability problems in first-order logic. Although this mapping does not cover the complete OCL language, our experience shows that the kind of OCL expressions typically used for specifying invariants and authorization constraints, are covered by the mapping. More intriguing is, however, the issue about the effectiveness of SMT solvers for automatically reasoning about FGAC policies. Although our experience so far is extremely encouraging (all problems are solved in less than a second), we should not forget that our results completely depend on the interaction between (i) the way our mapping translates into first-order logic the relevant OCL expressions (invariants and authorization constraints), and (ii) the heuristics implemented in the SMT solver.

3.4.1 Motivation

Model-driven engineering supports the development of complex software systems by generating software from models. Model-driven security [9] is a specialization of this paradigm, where system designs are modeled together with their security requirements, and security infrastructures are directly generated from the models. Of course, the quality of the generated code depends on the quality of the source models. If the models do not properly specify the system's intended behavior, one should not expect the generated system to do so either. *Quod natura non dat, Salmantica non praestat.*[9] Experience shows that even when using powerful, high-level modeling languages, it is easy to make logical errors and omissions. It is critical not only that the modeling language has a well-defined semantics, but also that there is tool support for analyzing the modeled systems' properties.

A running example

Here we introduce the example that we use to illustrate our methodology for reasoning about FGAC policies.

Data model. In Figure 3.3 we use ComponentUML's graphical syntax to define a data model, named `EmplBasic.dtm`. This model specifies that every employee may have a name, a surname, and a salary; that every employee may have a supervisor and may in turn supervise other employees; and that every employee may take one of two roles: Worker or Supervisor. In the terminology of ComponentUML, `Employee` is an *entity*; name, surname, salary, and role are *attributes*; supervisedBy and supervises are *association-ends*; and `Role` is an *enumerated class*. Notice that the association-end supervises has multiplicity `0..*`, meaning that an employee may supervise zero or more employees, while the association-end supervisedBy has multiplicity `0..1` meaning that an employee may have at most one supervisor.

Invariants We can refine the model `EmplBasic.dtm` (Figure 3.3) by adding invariants to this model. In particular, consider the following constraints:

1. There is exactly one employee who has no supervisor.

[9] Less elegantly said, *garbage in, garbage out.*

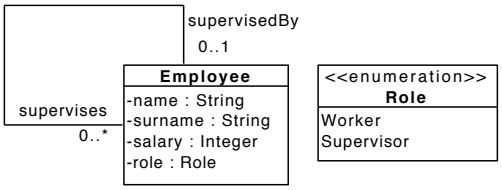


Figure 3.3: `EmplBasic.dtm`: a data model for employees' information.

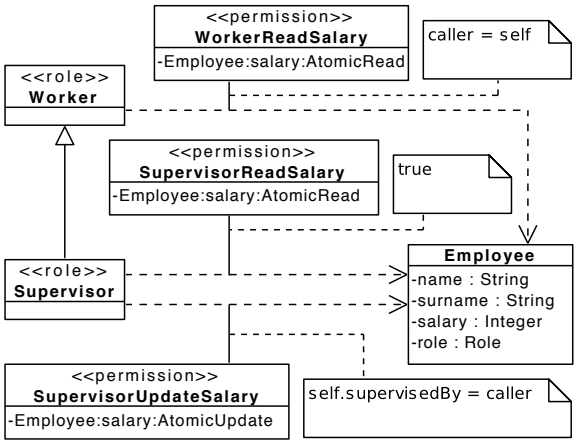


Figure 3.4: `Empl.stm`: a security model for accessing employees' information.

2. Nobody is its own supervisor.
3. An employee has role Supervisor if and only if it has at least one supervisee.
4. Every employee has one role.

These constraints can be formalized in OCL as follows:

(1) Employee.allInstances()−>one(e|e.supervisedBy.oclIsUndefined())
(2) Employee.allInstances()−>forAll(e|not(e.supervisedBy = e))
(3) Employee.allInstances()−>forAll(e|(e.role = Supervisor implies e.supervises−>notEmpty())
and (e.supervises−>notEmpty() implies e.role = Supervisor))
(4) Employee.allInstances()−>forAll(e|not(e.role.oclIsUndefined())

In what follows, we will refer to the constraint (1) as `oneBoss`, (2) as `noSelfSuper`, (3) as `roleSuper`, and (4) as `allRole`. Also, we will denote by `Empl1.dtm` the refined version of `EmplBasic.dtm` that includes as invariants the constraints `oneBoss`, `noSelfSuper`, `roleSuper`, and `allRole`.

Security model. In Figure 3.4 we use SecureUML's graphical syntax to define a security model, named `Empl.stm`. This model specifies a basic FGAC policy for accessing the employees' information modeled in `Empl1.dtm`. Permissions are assigned to users depending on their *roles*, which can be Worker or Supervisor. Also, users with role Supervisor *inherit* all the permissions granted to users with role Worker, since Supervisor is a *subrole* of Worker. Finally, permissions are constrained by *authorization constraints*: namely,

1. A worker is granted permission to read an employee's salary, provided that it is its own salary, as specified by the authorization constraint caller = self.

2. A supervisor is granted unrestricted permission to read an employee's salary, as specified by the authorization constraint true.

3. A supervisor is granted permission to update an employee's salary, provided that it supervises this employee, as specified by the authorization constraint self.supervisedBy = caller.

Recall the function *Auth* discussed in Section 2.1.3: namely, that for every security model S, role r, and action act, Auth($\mathcal{S}, r, act$) returns the authorization constraint defined in $\mathcal{S}$ for r to execute act. Consider now the value of Auth($\mathcal{S}, r, act$) in the following cases:

Auth(`Empl.stm`, Worker, update(salary))= false.

Auth(`Empl.stm`, Supervisor, update(salary))=
self.supervisedBy = caller or false.

Auth(`Empl.stm`, Worker, read(salary))= caller = self.

Auth(`Empl.stm`, Supervisor, read(salary))= caller = self or true.

3.4.2 Categories of security properties

Next, we will explain and illustrate with examples, how one can use the mapping from OCL to first-order logic proposed in [39], to reason about security models. This mapping essentially consists of two, inter-related components: (i) a map from data models and Boolean OCL expressions to first-order formulas, called ocl2fol$_{\text{def}}$; and (ii) a map from Boolean OCL expressions to first-order formulas, called ocl2fol. The following remark formalizes the main property of this mapping:

Remark 2 *Let $\mathcal{D}$ be a data model, with invariants $expr_1, \ldots, expr_n$, and let expr be a Boolean expression. Then, expr evaluates to* `true` *in every valid instance of $\mathcal{D}$ if and only if*

$$\begin{aligned}&\text{ocl2fol}_{\text{def}}(\mathcal{D})\\ &\quad\cup\bigcup_{i=1}^{n}\text{ocl2fol}_{\text{def}}(expr_i)\cup\bigcup_{i=1}^{n}\{\text{ocl2fol}(expr_i)\}\\ &\quad\cup\ \text{ocl2fol}_{\text{def}}(expr)\cup\{\neg(\text{ocl2fol}(expr))\}\end{aligned}$$

is unsatisfiable.

In what follows, given a security model $\mathcal{S}$, we use the term *scenario* to refer to any valid instance of $\mathcal{S}$'s underlying data model in which a user requests permission to execute an action. For the sake of simplicity, we will assume that neither the user requesting permission, nor the resource upon which the action will be executed, can be *undefined*.

We organize our examples in blocks or categories. In the first block, we are interested in knowing if there is any scenario in which someone with role r will be allowed to execute an action act. Notice that, by Remark 2 the answer will be 'No' if and only if the following set of formulas is unsatisfiable:

$$\begin{aligned}&\text{ocl2fol}_{\text{def}}(\mathcal{D})\cup\ \{\exists(caller)\exists(self)\exists(target)\exists(value)\\ &\ (\text{ocl2fol}(caller.\mathsf{role}=\mathsf{r})\wedge\text{ocl2fol}(\text{Auth}(\mathcal{S},r,act)))\}.\end{aligned}$$

Example 8 Consider the following question: *Is there any scenario in which someone with role Worker is allowed to change the salary of someone else (including itself)?* Recall that

Auth(`Empl.stm`, Worker, update(salary))= false.

According to Remark 2, the answer to this question is 'No', since the following set of formulas is clearly unsatisfiable:

$$\begin{aligned}&\text{ocl2fol}_{\text{def}}(\texttt{Empl1.dtm}) \cup \{\exists(\mathit{caller})\exists(\mathit{self})\\&\quad(\text{ocl2fol}(\mathit{caller}.\mathsf{role} = \mathsf{Worker}) \wedge \text{ocl2fol}(\mathsf{false}))\}.\end{aligned}$$

Example 9 Consider the following question: *Is there any scenario in which someone with role Supervisor is allowed to change the salary of someone else (including itself)?* Recall that

Auth(`Empl.stm`, Supervisor, update(salary))= (self.supervisedBy = *caller* or false).

According to Remark 2, the answer to this question is 'Yes', since the following set of formulas is satisfiable:

$$\begin{aligned}&\text{ocl2fol}_{\text{def}}(\texttt{Empl1.dtm}) \cup \{\exists(\mathit{caller})\exists(\mathit{self})\\&\quad(\text{ocl2fol}(\mathit{caller}.\mathsf{role} = \mathsf{Supervisor})\\&\quad\wedge \text{ocl2fol}(\mathit{self}.\mathsf{supervisedBy} = \mathit{caller}\ \mathsf{or}\ \mathsf{false}))\}.\end{aligned}$$

Example 10 Consider the following question: *Is there any scenario in which someone with role Supervisor is allowed to change its own salary?* Notice that in any scenario in which someone is requesting to change its own salary, the values of *self* (i.e., the employee whose salary is to be updated) and *caller* (i.e., the employee who is updating this salary) are the same. According to Remark 2, the answer to this question is 'No', since the following set of formulas is unsatisfiable:

$$\begin{aligned}&\text{ocl2fol}_{\text{def}}(\texttt{Empl1.dtm}) \cup \{\exists(\mathit{caller})\exists(\mathit{self})\\&\quad(\text{ocl2fol}(\mathit{caller}.\mathsf{role} = \mathsf{Supervisor})\\&\quad\wedge \text{ocl2fol}(\mathit{self} = \mathit{caller}\ \mathsf{and}(\mathit{self}.\mathsf{supervisedBy} = \mathit{caller}\ \mathsf{or}\ \mathsf{false})))\}.\end{aligned}$$

Example 11 Consider the following question: *Is there any scenario in which someone with role Supervisor is allowed to change the salary of someone who has no supervisor at all?* Notice that in any scenario in which someone (*caller*) is requesting to change the salary of someone (*self*) who has no supervisor at all, the value of *self*.supervisedBy must be null. According to Remark 2, the answer to this question is 'No', since the following set of formulas is unsatisfiable:

$$\begin{aligned}&\text{ocl2fol}_{\text{def}}(\texttt{Empl1.dtm}) \cup \{\exists(\mathit{caller})\exists(\mathit{self})\\&\quad(\text{ocl2fol}(\mathit{caller}.\mathsf{role} = \mathsf{Supervisor})\\&\quad\wedge \text{ocl2fol}(\mathit{self}.\mathsf{supervisedBy} = \mathsf{null}\ \mathsf{and}\\&\qquad(\mathit{self}.\mathsf{supervisedBy} = \mathit{caller}\ \mathsf{or}\ \mathsf{false})))\}.\end{aligned}$$

In our second block of examples, we are interested in knowing if there is any scenario in which someone with role r will not be allowed to execute an action act. Notice that, by Remark 2, the answer will be 'No' if and only if the following set of formulas is unsatisfiable:

$$\begin{aligned}&\text{ocl2fol}_{\text{def}}(\mathcal{D}) \cup \{\exists(\mathit{caller})\exists(\mathit{self})\exists(\mathit{target})\exists(\mathit{value})\\&\quad(\text{ocl2fol}(\mathit{caller}.\mathsf{role} = r) \wedge \neg(\text{Auth}(\mathcal{S}, r, \mathit{act})))\}.\end{aligned}$$

Example 12 Consider the following question: *Is there any scenario in which someone with role Supervisor is not allowed to change the salary of someone else (including itself)?* According to Remark 2, the answer to this question is 'Yes', since the following set of formulas is satisfiable:

$$\begin{aligned}&\text{ocl2fol}_{\text{def}}(\texttt{Empl1.dtm}) \cup \{\exists(\mathit{caller})\exists(\mathit{self})\\&(\text{ocl2fol}(\mathit{caller}.\mathsf{role} = \mathsf{Supervisor}) \wedge\\&\neg(\text{ocl2fol}(\mathit{self}.\mathsf{supervisedBy} = \mathit{caller}\ \mathsf{or}\ \mathsf{false})))\}.\end{aligned}$$

In our third block of examples, we are interested in knowing if there is any scenario in which nobody with role r will be allowed to execute an action *act*. Notice that, by Remark 2, the answer will be 'No' if and only if the following set of formulas is unsatisfiable:

$$\begin{aligned}&\text{ocl2fol}_{\text{def}}(\mathcal{D}) \cup \{\exists(\mathit{self})\exists(\mathit{target})\exists(\mathit{value})\forall(\mathit{caller})\\&(\text{ocl2fol}(\mathit{caller}.\mathsf{role} = r) \Rightarrow\\&\neg(\text{ocl2fol}(\text{Auth}(\mathcal{S}, r, \mathit{act}))))\}.\end{aligned}$$

Example 13 Consider the following question: *Is there any scenario in which nobody with role Supervisor is allowed to change the salary of someone else (including itself)*? According to Remark 2, the answer to this question is 'Yes', since the following set of formulas, which we will refer to as Forms(Ex 13), is satisfiable:

$$\begin{aligned}&\text{ocl2fol}_{\text{def}}(\texttt{Empl1.dtm}) \cup \{\exists(\mathit{self})\forall(\mathit{caller})\\&(\text{ocl2fol}(\mathit{caller}.\mathsf{role} = \mathsf{Supervisor}) \Rightarrow\\&\neg(\text{ocl2fol}(\mathit{self}.\mathsf{supervisedBy} = \mathit{caller}\ \mathsf{or}\ \mathsf{false})))\}.\end{aligned}$$

In our fourth block of examples, we are interested in knowing if, in every scenario, there is at least one object upon which nobody with role r will be allowed to execute an action *act*. Notice that, by Remark 2, the answer will be 'Yes' if and only if the following set of formulas is unsatisfiable:

$$\begin{aligned}&\text{ocl2fol}_{\text{def}}(\mathcal{D}) \cup \{\forall(\mathit{self})\exists(\mathit{target})\exists(\mathit{value})\exists(\mathit{caller})\\&(\text{ocl2fol}(\mathit{caller}.\mathsf{role} = r) \wedge \text{ocl2fol}(\text{Auth}(\mathcal{S}, r, \mathit{act})))\}.\end{aligned}$$

Example 14 Consider the following question: *In every scenario, is there at least one employee whose salary can not be changed by anybody with role Supervisor?* According to Remark 2, the answer to this question is 'Yes', since the following set of formulas is unsatisfiable:

$$\begin{aligned}&\text{ocl2fol}_{\text{def}}(\texttt{Empl1.dtm}) \cup \{\forall(\mathit{self})\exists(\mathit{caller})\\&(\text{ocl2fol}(\mathit{caller}.\mathsf{role} = \mathsf{Supervisor}) \wedge\\&\text{ocl2fol}(\mathit{self}.\mathsf{supervisedBy} = \mathit{caller}\ \mathsf{or}\ \mathsf{false}))\}.\end{aligned}$$

To conclude, we want to illustrate the importance of taking into account the invariants of the underlying data model when reasoning about FGAC policies. Let `Empl2.dtm` be the data model that results from adding to the model `EmplBasic.dtm` the invariants `noSelfSuper`, `roleSuper`, `allRole`, plus the following invariant:

5. Everybody has one supervisor.

This invariant, which we will refer to as `allSuper`, can be formalized in OCL as follows:
Employee.allInstances()−>forAll(e|not(e.supervisedBy.oclIsUndefined()))

Example 15 Consider the security model `Empl.stm` this time with `Empl2.dtm` as its underlying data model. Consider again the question that we asked ourselves in Example 13: namely, *is there any scenario in which nobody with role Supervisor is allowed to change the salary of someone else (including itself)*? According to Remark 2, the answer to this question is different from Example 13, namely, 'No', since the set of formulas Forms(Ex 13) is now unsatisfiable.

Finally, let `Empl3.dtm` be the data model that results from removing from `Empl2.dtm`, the invariant `roleSuper`.

Example 16 Consider the security model `Empl.stm`, but this time with `Empl3.dtm` as its underlying data model. Consider, once again, the question that we asked ourselves in Example 13: namely, *is there any scenario in which nobody with role Supervisor is allowed to change the salary of someone else (including itself)*? According to Remark 2, the answer to this question is now different from Example 15, namely, 'Yes', since the set of formulas Forms(Ex 13) is now satisfiable.

3.4.3 Proving security properties

We briefly report here on our experience using the Z3 SMT solver [45] to automatically obtain the answers to the questions posed in the examples in Section 3.4.2. Table 3.7 below summarizes the results of our experiments. For each example, we show the time that Z3 takes to return an answer (in all cases, less than 1 second); the answer that it returns (in all cases, the expected one); and the first-order model that it generates when the answer is `sat`, i.e., when it finds that the input set of formulas is satisfiable. Each model represents a scenario (not necessarily the one discussed in Section 3.4.2 for the corresponding example), and here we simply indicate the number of employees that it contains, which employees are linked through the association-end `supervisedBy`, which employees have the role `Worker`, which employees have the role `Supervisor`, which employee is the one requesting permission to change the salary (*caller*), and which employee is the one whose salary will be changed (*self*), if permission is granted. We ran our experiments on a laptop computer, with a 2.66GHz Intel Core 2 Duo processor and 4GB 1067 MHz memory, using Z3 version 4.3.2 9d221c037a95-x64-osx-10.9.2. Finally, the input for Z3 has been generated using our tool SecProver [79]. This tool takes the following parameters: a data model, a security model, a set (possibly empty) of invariants, an action, a role, a set (possibly empty) of additional constraints, and a *question type*. SecProver automatically generates the set of first-order formulas whose satisfiability will determine, according to our methodology, the answer to the given question.

Ex.	Time	Ans.	Interpretation
8	0.078s	`unsat`	—

9	0.107s	`sat`	#employees = 3
			supervisedBy = {(e_3, e_2),
			(e_1, e_2)}
			Worker = {e_1, e_3}
			Supervisor = {e_2}
			caller = e_2, *self* = e_1

10	0.041s	`unsat`	—
11	0.042s	`unsat`	—

12	0.306s	`sat`	#employees = 6
			supervisedBy = {(e_1, e_2),
			(e_2, e_3), (e_4, e_2),
			(e_5, e_3), (e_6, e_3)}
			Worker = {e_1, e_4, e_5, e_6}
			Supervisor = {e_2, e_3}
			caller = e_3, *self* = e_1

13	0.078s	`sat`	#employees = 1
			supervisedBy = ∅
			Worker = {e_1}
			Supervisor = ∅
			self = e_1

14	0.485s	`unsat`	—
15	0.060s	`unsat`	—

16	0.506s	`sat`	#employees = 15
			supervisedBy = {(e_1, e_2),
			(e_2, e_4), (e_3, e_4), (e_4, e_6),
			(e_5, e_4)(e_6, e_{12}), (e_7, e_4),
			(e_8, e_{14}), (e_9, e_4), (e_{10}, e_4),
			(e_{11}, e_{15}), (e_{12}, e_{13}),
			(e_{13}, e_4), (e_{14}, e_4), (e_{15}, e_4)}
			Worker = all
			Supervisor = ∅
			self = e_2

Table 3.7: Automatic reasoning over the examples 8-16 introduced in Section 3.4.2.

Chapter 4

Validating model-driven development

To provide evidence of the usefulness of our methodology, we report in this Chapter on our use of ActionGUI to develop a secure data-management application, based on an eHealth case study proposed within NESSoS, the European Network of Excellence on Engineering Secure Future Internet Software Services and Systems [71]. The NESSoS eHealth case study consists of a web-based system for electronic health record management (EHRM). Electronic health records (EHR) store information created by, or on behalf of, a health professional in the context of the care of a patient. As further evidence of the usefulness of our methodology, we also include in this Chapter a summary of four other web applications that we have developed using ActionGUI.

Overall, our experience demonstrates the methodology's potential for developing real-world applications. First, the use of model-transformation and code-generation frees the developer from programming fine-grained authorization constraints and inserting them at all the required places throughout the application's code and with the correct arguments. Except for small applications, this is cumbersome and error-prone, since the number of data actions associated to events may be on the order of hundreds. Secondly, our methodology supports modularity and separation of concerns. In particular, the security model can be changed independently of the GUI model, without requiring one to re-program and re-insert all the new fine-grained authorization constraints, since this is automatically done by our model-transformation.

4.1 A secure eHealth application

We report here on our use of ActionGUI to develop a secure data-management application, called eHRMApp. This application is based on a case study proposed within NESSoS, the European Network of Excellence on Engineering Secure Future Internet Software Services and Systems [71]. The eHealth case study consists of a web-based system for electronic health record management (EHRM). Electronic health records (EHR) store information created by, or on behalf of, a health professional in the context of the care of a patient.

Electronic health records are highly sensitive and therefore their access must be controlled. Part of the challenge in this case study was to model the access control policy and build an application that enforces it at runtime. The policy consists of various authorization rules along the lines of: *The access control criteria for an EHR depends, among others, on the type of EHR. For instance, a highly sensitive record might be only avail-*

able to the patient's treating doctor (and perhaps a few others, in rare situations). Such rules necessitate fine-grained access control, where access control decisions depend not only on the user's credentials but also on the satisfaction of constraints on the state of the persistence layer, i.e. on the values of stored data items.

We show how ActionGUI's modeling languages can be used to specify the application's data model (e.g., hospital staff, health records), security policy (e.g., rules like the above) and behavior. Moreover, by illustrative examples, we highlight various features of the ActionGUI methodology and toolkit. Overall, the eHealth case study is interesting as an example of developing a secure data-management application, and it provides a proof-of-concept for the application of the ActionGUI methodology to an industry-relevant problem.

The NESSoS eHRMApp application scenario defines different system use cases along with the associated access control policy. The use cases include: register new patients in a hospital and assign them to clinicians, such as nurses or doctors; retrieve patient's information; register new nurses and doctors in a hospital and assign them to a ward; change nurses or doctors from one ward to another; and reassign patients to doctors. We will focus on a representative use case as a running example: reassigning patients to doctors. We will use this example to illustrate ActionGUI's modeling languages as well as the model-based separation of concerns supported by the ActionGUI methodology.

4.1.1 The eHRMApp's data model

The full data model for the eHRMApp application contains 18 entities, 40 attributes, and 48 association-ends. We discuss below just the entities, attributes, and association-ends that are required for our running example.

Figure 4.1 presents this data model, formalized using ComponentUML's textual syntax. As this example shows, ActionGUI data models specify how the application's data is structured, independently of how it will be visualized or accessed.

Professional. This entity represents the eHRMApp's users. The role assigned to each user is specified by its role attribute. The roles considered are DIRECTOR, ADMINISTRATOR, DOCTOR, NURSE, and SYSTEM. The medical centers where a user works are linked to the user through the association-end worksIn. If a user is a doctor, then it is linked to the corresponding doctor information through the association-end asDoctor. Similarly, if a user is an administrative staff, then it is linked to staff information through the association-end asAdministrative.

MedicalCenter. This entity represents medical centers. The departments belonging to a medical center are linked to the medical center through the association-end departments. The professionals working for a medical center are linked to the medical center through the association-end employees.

Doctor. This entity represents doctor's information. Doctor's information is linked to the corresponding professional through the association-end doctorProfessional. The departments where a doctor works are linked to the doctor's information through the association-end doctorDepartments. The patients treated by a doctor are linked to the doctor's information through the association-end doctorPatients.

```
Entity Professional {
  Role role
  Set(MedicalCenter) worksIn oppositeTo employees
  Doctor asDoctor oppositeTo doctorProfessional
  Administrative asAdministrative oppositeTo administrativeProfessional  }
Entity MedicalCenter {
  Set(Department) departments oppositeTo belongsTo
  Set(Professional) employees oppositeTo worksIn  }
Entity Doctor {
  Professional doctorProfessional oppositeTo asDoctor
  Set(Department) doctorDepartments oppositeTo doctors
  Set(Patient) doctorPatients oppositeTo doctor  }
Entity Administrative {
  Professional administrativeProfessional oppositeTo asAdministrative  }
Entity Department {
  MedicalCenter belongsTo oppositeTo departments
  Set(Doctor) doctors oppositeTo doctorDepartments
  Set(Patient) patients oppositeTo department  }
Entity Patient {
  Doctor doctor oppositeTo doctorPatients
  Department department oppositeTo patients  }
enum Role { DIRECTOR ADMINISTRATOR DOCTOR NURSE SYSTEM }
```

Figure 4.1: The eHRMApp's data model (partial).

Administrative. This entity represents administrative staff information. Administrative staff information is linked to the corresponding professional through the association-end administrativeProfessional.

Department. This entity represents departments. The medical center to which a department belongs is linked to the department through the association-end belongsTo. The doctors working in a department are linked to the department through the association-end doctors. The patients treated in a department are linked to the department through the association-end patients.

Patient. This entity represents patients. The doctor treating a patient is linked to the patient through the association-end doctor. The department where a patient is treated is linked to the patient through the association-end department.

4.1.2 The eHRMApp data model's invariants

The full eHRMApp application data model is constrained by 66 data invariants, formalized using OCL. The following three invariants are representative.

1. *Each patient is treated by a doctor.*
   Patient.allInstances()→forAll(p|not(p.doctor.oclIsUndefined()))
2. *Each patient is treated in a department.*
   Patient.allInstances()→forAll(p|not(p.department.oclIsUndefined()))
3. *Each patient is treated by a doctor who works for a set of departments, including the department where the patient is treated.*
   Patient.allInstances()→forAll(p| p.doctor.doctorDepartments→includes(p.department))

```
1  Role ADMINISTRATOR {
2    Patient{
3      if caller.worksIn→includes(value.belongsTo) then Update::department }
4    Patient{
5      if caller.worksIn→exists(m|value.doctorProfessional.worksIn→includes(m))
6           and caller.worksIn→includes(self.department.belongsTo)
7        then Update::doctor }
```

Figure 4.2: Examples of the eHRMApp security model's permissions.

These invariants make precise the intended meaning of the associations between the entities Patient, Doctor, and Department. The first two invariants state that the doctor and the department associated to a patient cannot be undefined, i.e., *null*. The third invariant states that a doctor who treats a patient must work in the department where the patient is treated, although the doctor may also work in other departments.

4.1.3 The eHRMApp's security model

Electronic health records are, by their nature, highly sensitive and the NESSoS case study informally defines the policy that regulates their access. As expected, the authorization to carry out certain actions is not only role-based, but also context-based. In other words, the eHRMApp access control policy is *fine-grained*.

The full eHRMApp application's security model contains 5 roles and 573 permissions, where each permission authorizes users in a role, to execute an action upon the satisfaction of an authorization constraint formalized in OCL. In Figure 4.2 we present examples of two permissions, modeled using SecureUML's textual syntax. The first permission authorizes a user (*caller*) with the role ADMINISTRATOR, to reassign a patient (self) to a department (*value*), provided that the user works in a set of medical centers, that includes the one to which the department will be reassigned. The second permission authorizes a user (*caller*) with the role ADMINISTRATOR, to reassign a patient (*self*) to a doctor (*value*), provided two conditions are satisfied: (i) the user works in a medical center, where the doctor to which the patient will be reassigned, also works; and (ii) the user works in a medical center, that owns the department where the patient is currently being treated.

As this example illustrates, ActionGUI's security models are formulated in terms of the application's data. This formalization is independent of how the data is visualized or accessed through the application's graphical user interface.

4.1.4 The eHRMApp's GUI model

The full eHRMApp application's GUI model contains 8 windows for the following use cases: login to the application; access a medical center's information; register a new patient; review a patient's information; reassign a patient to a doctor and department; access options reserved for the medical center's director; introduce a professional into the system; and reassign a professional to a department. Here are some other concrete figures about the size of the GUI model: i) Widgets: 19 buttons; 73 labels; 19 text fields; 5 boolean fields; 1 date field; 1 combo-box; and 9 tables; ii) Statements: 34 if-then-else statements; iii) Data actions: 11 create actions; 41 update actions; 5 add link actions; and 2 remove link actions; iv) GUI actions: 157 set actions; and 7 open actions; v) OCL expressions: 361 expressions (77 non-literals).

We discuss below the window relevant for our running example: the window movePatientWI for reassigning a patient to a doctor and a department. Figures 4.3 and 4.4 present our model of this window, in GUIML's textual syntax. Note that, in contrast with Example 4, **onCreate** events are implicitly declared in widget variables' initialization. Figure 4.5 contains a screenshot of the actual window generated from this model. The window movePatientWI assumes that both a medical center and a patient have previously been selected. This information is stored, respectively, in the variables medicalCenter and patient (lines 2-3). The window movePatientWI contains the following widgets:

- A label patientLa that displays the name and surname of the selected patient (lines 5–7).
- A label departmentLa that displays the name of the department where the selected patient is treated (lines 8–9).
- A label doctorLa that displays the name and surname of the doctor who treats the selected patient (lines 10–13).
- A label departmentsLa that displays a message inviting the user to select a department (lines 14–15).
- A label doctorsLa that displays a message inviting the user to select a doctor (lines 16–17).
- A table departmentsTa that displays information about the departments that belong to the selected medical center (line 22); in particular, the name of each of these departments is shown (line 31-34). Also, when the user selects a department from this list, it refreshes the list of doctors displayed in the table doctorsTa (see below) with the doctors who work for the selected department (lines 19–21).
- A table doctorsTA that is initially empty (line 24). As previously explained, upon selection of a department in the table departmentsTa, it displays information about the doctors who work for the selected department (lines 19–21); in particular, the name and surname of each of these doctors are shown (lines 35-41).
- A button moveBu that, when clicked upon, if there is a department selected in the table departmentsTa (line 44), and there is also a doctor selected in the table doctorsTa (line 45), then:
  - it reassigns the selected department to the selected patient (line 46);
  - it reassigns the selected doctor to the selected patient (line 47);
  - it notifies the user that the reassignment succeeded (lines 48).

  Otherwise, it notifies the user that either a doctor (line 50) or a department (line 52) must first be selected.
- A button backBU that, when the user clicks on it, it returns to the previous window (line 55).

As this example illustrates, ActionGUI's GUI models depend on how the application's data is structured — after all, they describe how users interact with this data — but not on the application's security policy. Of course, in terms of the final application's *usability*, there is a dependency: a GUI can end up being unusable precisely because of the application's security policy.

```
1  Window movePatientWi {
2    MedicalCenter medicalCenter
3    Patient patient
4    String text := ['Move a patient']

5  Label patientLa {
6    String text := ['Patient: '.concat($movePatientWi.patient$.contact.name)
7                    .concat(' ').concat($movePatientWi.patient$.contact.surname)] }
8  Label departmentLa {
9    String text := ['Department: '.concat($movePatientWi.patient$.department.name)] }

10 Label doctorLa {
11   String text := ['Doctor: '.concat($movePatientWi.patient$.doctor.
12                   doctorProfessional.name).concat(' ').
13                   concat($movePatientWi.patient$.doctor.doctorProfessional.surname)] }

14 Label departmentsLa {
15   String text := ['Select the new department:'] }

16 Label doctorsLa {
17   String text := ['Select the new doctor:'] }

18 Table departmentsTa {
19   Department selected {
20     if [not $selected$.oclIsUndefined()] {
21       movePatientWi.doctorsTa.rows := [$selected$.doctors] } }
22   Set(Department) rows := [$movePatientWi.medicalCenter$.departments] }

23 Table doctorsTa {
24   Set(Doctor) rows := [Doctor.allInstances()→select(false)]
25   Doctor selected }

26 Button moveBu {
27   String text := ['Move the patient'] }

28 Button backBu {
29   String text := ['Back']
30 }
```

Figure 4.3: A window for reassigning a selected patient (part I)

```
31 Table movePatientWi.departmentsTa {
32   columns{
33           ['Department'] : Label department {
34                               String text := [$departmentsTa.row$.name] } } }

35 Table movePatientWi.doctorsTa {
36   columns {
37           ['Doctor'] : Label doctor {
38                          String text :=
39                            [$doctorsTa.row$.doctorProfessional.name
40                              .concat(' ')
41                              .concat($doctorsTa.row$.doctorProfessional.surname)] } } }

42 Button movePatientWi.moveBu {
43   event onClick {
44         if [not $departmentsTa.selected$.oclIsUndefined()] {
45           if[not $doctorsTa.selected$.oclIsUndefined()] {
46            [$movePatientWi.patient$.department] := [$departmentsTa.selected$]
47            [$movePatientWi.patient$.doctor] := [$doctorsTa.selected$]
48            notification(['Success'],['The patient has been reassigned.'],[0]) }
49           else {
50            notification(['Error'],['Please, select first a doctor.'],[0]) } }
51         else {
52           notification(['Error'],['Please, select first a department.'],[0]) } } } }

53 Button movePatientWi.backBu {
54   event onClick {
55         back } }
```

Figure 4.4: A window for reassigning a selected patient (part II)

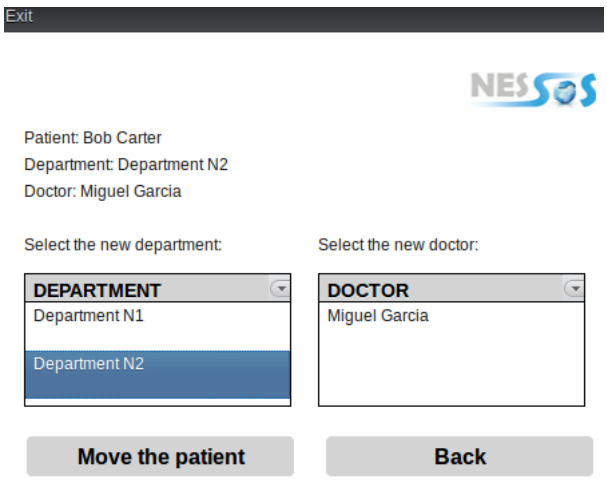


Figure 4.5: Screenshot of the window for reassigning a selected patient

```
46.1  if [[$movePatientWi.role$ = ADMINISTRATOR
46.2        and $movePatientWi.caller$.worksIn
46.3              →includes($departmentsTa.selected$.belongsTo)] {
46.4     [$movePatientWi.patient$.department] := [$departmentsTa.selected$] }
46.5     else { fail }

47.1  if [[$movePatientWi.role$ = ADMINISTRATOR
47.2        and $movePatientWi.caller$.worksIn→exists(m|
47.3              $doctorsTa.selected$.doctorProfessional.worksIn→includes(m))
47.4            and $movePatientWi.caller$.worksIn
47.5              →includes($movePatientWi.patient$.department.belongsTo))]  {
47.6     [$movePatientWi.patient$.doctor] := [$doctorsTa.selected$] }
47.7     else { fail }
```

Figure 4.6: The security-aware actions for reassigning a selected patient

4.1.5 The eHRMApp's security-aware GUI model

As explained in Section 2.3, the heart of ActionGUI is a model-transformation function that, essentially, prefixes each data action in the GUI model with the authorization check specified in the security model. The full eHRMApp application's GUI model contains 59 data actions, and therefore the automatically generated eHRMApp application's security-aware GUI model contains the same number of authorization checks.

As an illustrative example of our model-transformation function, we show in Figure 4.6 the part of the security-aware GUI model for the button moveBu's event **onClick** that is relevant for our running example. The action of reassigning the selected patient to the department selected in the table departmentsTa (line 46 in Figure 4.4) is now wrapped by an if-then-else statement (lines 46.1-46.5 in Figure 4.6) whose condition reflects the permission for executing this action given by line 3 in Figure 4.2. Similarly, the action of reassigning the selected patient to the doctor selected in the table doctorsTa (line 47 in Figure 4.4) is wrapped by an if-then-else statement (lines 47.1-47.7 in Figure 4.6) whose condition reflects the permission for executing this action given by lines 5–7 in Figure 4.2.

4.1.6 Generating the eHRMApp application

The ActionGUI toolkit automatically generates the complete eHRMApp application in under 10 seconds. The generated .war file includes the Vaadin library as well as other external libraries. The Vaadin library is responsible of 70% of the size of the generated file and only 10% of this file corresponds to the code that ActionGUI automatically generates to interpret the application's model. The size of the .war file containing the complete application is roughly 12 MB.

4.2 Other applications

We report here on other four web applications that we developed using ActionGUI. We begin by briefly describing these applications and then we provide concrete figures, which include also the eHRMApp application extensively described before, using different metrics. We conclude with other more subjective considerations.

Customer Relationship Management (CRMApp) We have developed a web application for managing customers of a Hospital and Care Center. This application allows marketing and public relations personnel to manage contact information, including filtering contacts based on different criteria and exporting the results in Excel files. As customer data is highly sensitive, data is subject to a restrictive access-control policy. For example, a marketing and PR staff member can only access the contact information, of those contacts previously selected as targets of a marketing campaign to which he is assigned. The application also allows a General Manager to create marketing campaigns, select the targeted patients, and assign marketing and PR staff members to campaigns.

Volunteer Management (VMApp) We have developed a web application for managing a care center's volunteer program. Using this application, the program's coordinators can take actions such as: introduce new volunteers; create, edit, and modify tasks; and propose these tasks to the volunteers, based on the volunteers' time availability and preferences. The access-control policy stipulates, for example, that volunteers are only authorized to edit their own personal information, such as their preferences and time availability, and to accept or reject their own tasks.

Meal Service Management (MSMApp) This is a web application for managing a student residence's meal service. Using this application, a resident can notify the administration whether he will have a meal at the residence's cafeteria, in which of the available time slots, and if he will bring a guest. A resident shall only edit his own meal selection and within a specific time window, which depends on the selected meal. Administrators can create new resident accounts, and list the meals requested for each available time slot.

Chatroom (ChatApp) This is an extension of our running example: in addition to posting messages in a selected chatrooms, users can also create and delete chatrooms, under specific conditions.

CRMApp, VMApp, and MSMApp are *commercial* applications. They were developed for real customers, and they are currently being used by their different stakeholders. In contrast, eHRMApp was developed, as we explain in Section 4.1, as part of a case study proposed by industrial partners in an ongoing European project. The interested reader can find more information about this case study, as well as a demo version of the eHRMApp application, at [1].

In Table 4.1 we measure these applications, reporting on the size of the database that they manage and the number of registered users (at the time of writing this manuscript). In particular, for each commercial application, we indicate: the number of registered users; the current size of the associated database; the number of tables defined by its database schema; and the maximum number of rows currently contained in any of these tables. In Table 4.2 we measure our applications from a different perspective, namely, the size of their underlying models. In particular, for each application, we show: the size of the corresponding XML files; the number of their widgets (by types); the number of if-then-else and iterate statements associate to their events; the number of data and GUI actions (by types) also associated to their events; and the number of OCL expressions used to define the conditions in the if-then-else statements, the sources in the iterator statements, or the arguments in the data and GUI actions. We also indicate how many of these OCL expressions are non-literal expressions, as well as their average size and the average number of widget variables that they contain.

	CRMApp	VMApp	MSMApp	eHRMApp	ChatApp
Commercial/Academic (*)	Com.	Com.	Com.	Acad.	Acad.
Num. of users (**)	25	95	24	NA	NA
Size of database (**)	4.19 Mb	1.18 Mb	0.9 Mb	NA	NA
Num. of tables	125	52	16	46	7
Max. num. of rows (**)	4100	112	665	NA	NA

(*) "Commercial" means that this application is being used in a business environment by real users and stakeholders. (**) At the time of writing this manuscript.

Table 4.1: Example applications: size of the application's data

	CRMApp	VMApp	MSMApp	eHRMApp	ChatApp
Models					
Size of data model (XML)	33,9 Kb	17.7 Kb	3,5 Kb	8,9 Kb	1,5 Kb
Size of security model (XML)	24,4 Kb	26.7 Kb	1,4 Kb	18,5 Kb	5,8 Kb
Size of GUI model (XML)	1611,2 Kb	1538,7 Kb	309,7 Kb	136,7 Kb	29,2 Kb
Widgets					
Num. of windows	49	102	11	8	3
Num. of buttons	182	293	30	18	10
Num. of labels	691	697	83	66	7
Num. of text fields	159	169	10	19	4
Num. of boolean fields	67	9	0	5	1
Num. of date fields	14	16	2	1	0
Num. of combo boxes	52	33	24	1	0
Num. of tables	65	85	7	9	2
Statements					
Num. of if-then-else	650	334	150	35	7
Num. of iterate	66	13	0	0	1
Data actions					
Num. of creates (entity)	50	22	4	11	2
Num. of deletes (entity)	14	33	0	0	2
Num. of updates	268	180	15	25	4
Num. of creates (assoc)	111	66	3	21	4
Num. of deletes (assoc)	32	30	0	4	0
GUI actions					
Num. of sets	1840	1553	569	120	24
Num. of opens	164	234	18	7	7
OCL Expressions					
Num. of expressions	3847	3221	925	331	74
Num. of non-literal expressions	1478	1105	390	80	16
Avrg. size of non-literal expressions (*)	5,21	5,93	5,31	4,06	5,75
Avrg. num. of widget variables per non-literal expression	1,09	1,27	1,17	1,09	0,94

(*) "Size" is calculated counting the number of OCL operators, including literals.

Table 4.2: Example applications: size of the application's models

With respect to code-generation, the ActionGUI toolkit generates all our example applications in under a minute. The generated .war file includes the Vaadin library and some other external libraries (as well as, in the case CRMApp and VMApp, custom code for sending mails and exporting data). The Vaadin library is in fact responsible of 70% of the size of the generated file, while only 10% of this file corresponds to the code that ActionGUI automatically generates to interpret the application's model (as expected, the size of this *interpreter* does not vary much between applications). For our examples, the average size of the .war file containing each application is roughly 12 MB.

4.3 Evaluation

We conclude this Chapter with several more subjective considerations. As these are just based on our experience, their significance should not be overemphasized. Nevertheless they may give the reader an impression of what it is like to use ActionGUI.

First, the learning curve for ActionGUI appears moderate. The time required to learn

to model a secure data-management application depends on the modeler's familiarity with class diagrams, security diagrams and, above all, OCL. Assuming modest knowledge of UML, learning to model (non-toy) applications using ActionGUI takes less than 4 hours, which is arguably much less than what is required to learn enough of web technologies to be able to program (and correctly deploy) these applications. This estimate is based in part on experience teaching model-driven development of secure data-management applications to students at ETH Zurich and having them carry out projects using ActionGUI.

Second, the time needed to model a secure data-management application depends on the experience of the modeler (in particular, his command of OCL) and the complexity of the application (in particular, the number and the size of the OCL expressions used in the model). However, as a rule of thumb, modeling a *menu* window that contains 6 buttons that, when clicked upon, will open other windows, may take less than 30 minutes. In contrast, modeling a window that contains an *online form* with 10 text fields and a button that, when clicked upon, will first check that the entries are correctly filled and then will submit the form (i.e., update the database), may take one to two hours. And approximately the same time is required to model a *select&display* window that contains a table with 5 columns and a combo box, and that will change the information displayed in the table depending on the combo box selection. The take-home message is that modeling time is directly proportional to the number and size of the OCL expressions used in the models. This should not come as a surprise since these expressions are the ones that define the application's logic.

Finally, the main advantages of our approach concern the quality and ease of maintenance of the generated applications, which results from using our many-models-to-models transformation and our code-generator. First, this transformation effectively frees the developer from having to program fine-grained authorization constraints and insert them at all the required places within the application's code and with the correct arguments. Except for small applications, this is a cumbersome and error-prone task, since the number of data actions associated to events may easily be on the order of several hundred; see, for example, CRMApp and VMApp. Moreover, these data actions are typically called with different arguments each time. Second, our approach supports modularity and separation of concerns. In particular, the security model can be changed independently of the GUI model, without worrying about re-programming and re-inserting all the new fine-grained authorization constraints since this is automatically done by our transformation. Again, given the large number of these checks together with the complexity of the corresponding authorizations, we argue that, for non-toy security data-management applications, our approach effectively reduces the maintenance costs.

Chapter 5

Related work

In this Chapter, we focus on works that are directly related to the main contribution of our work, namely, ActionGUI as a novel, tool-supported model-driven methodology for developing secure data-management applications. We have organized the related work in two sections, not completely disjoint. First, we will compare ActionGUI with other proposals for modeling data-management applications, and, in particular, their graphical user interfaces. Then, we will compare ActionGUI with other tools (in this case, commercial tools) for developing secure data-management applications.

Modeling secure data-management applications As a modeling tool, UWE [14, 21, 19, 63] provides the modeler with a higher-level of abstraction than ActionGUI. In particular, the actions executed by the widgets' events are described in UWE using natural language. Thus, unless the models are appropriately refined, as discussed in [63], UWE does not support code-generation. In contrast, UWE provides specific diagrams for modeling GUI *presentations* and *navigations*, which facilitate the task of GUI modeling. In this respect, we define in [20] a mapping that transforms high-level UWE models into more concrete ActionGUI models that, once completed by the modeler, can be directly used to generate the intended applications. Finally, [19] extends UWE to use SecureUML for modeling security policies. However, this work does not use a model-transformation to lift the security policy to the GUI level. Instead the UWE modeler is responsible for adding all the appropriate authorization checks to the GUI model.

Like ActionGUI, ZOOM [60] allows GUI modelers to specify widgets, their events, and their actions. Moreover, using an extension of Z [86], one can specify the conditions of the actions and their arguments, similar to how this is done in ActionGUI using OCL. In contrast to ActionGUI, ZOOM does not provide a language for modeling security, and security aspects are not explicitly considered in this approach. Moreover, ZOOM does not support code-generation. It only provides interpreters for model animation.

In contrast to ActionGUI, UWE, and ZOOM, the approaches presented in [37, 38, 51] do not provide a language for modeling GUIs. They instead implement different *rules* for automatically deriving GUIs, based on either the application's data model, as in [37, 38], or the application's prototypical scenarios, as in [51]. As expected, the behavior of the resulting GUIs is limited and, based on our experience, insufficient to cope with the logic embedded in real data-management applications. Moreover, security aspects are not addressed in these proposals.

There is other related work that falls between the two extremes of full GUI modeling and full GUI derivation. The OO-method [74, 69] supports building GUIs using UI-*patterns*. These patterns specify the possible interactions with the application's data

based on the classes, attributes, and associations that are declared in the underlying data model. This approach has the advantage of reducing the time required for modeling GUIs. However, the UI-patterns impose restrictions on the type of GUIs that can be modeled, both in terms of their structure and their behavior. Moreover, this approach only supports role-based access control, but not fine-grained access control. The Interaction Flow Modeling Language (IFML) [85] is another approach that falls in this category. IFML is the OMG's standard modeling language for expressing content, user interaction and control behavior at the applications' front-ends, as well as the binding to the applications' persistence and business logic layers. IFML is an improved version of its predecessor, the WebML language [28]. Concretely, WebML was oriented to design component-based platform-specific models (PSMs), for web applications' front-ends; whereas IFML is oriented to design component-based platform-independent models (PIMs), for any kind of applications' front-ends. IFML is integrated in the commercial WebRatio toolkit [84]. Its component-based language design imposes, again, restrictions on the type of GUIs that can be modeled. Furthermore, security aspects are not at all considered in IFML.

Other approaches that fall in this category are [78, 64]. In both cases, the modeler must associate to each widget container the specific data type accessed using the widget. As before, the possible interactions with the underlying data is limited by the default behavior implemented for these widget containers. Security aspects are also not considered.

Finally, there are approaches whose primary focus is to support UI design at different levels of abstraction. Prominent examples are the XML User Interface Language (XUL) [70] and the USer Interface eXtensible Markup Language (UsiXML) [65]. XUL is the Mozilla's XML-based language for building user interfaces of applications like Firefox. UsiXML is an XML-compliant markup language that describes the UI for multiple usage contexts, such as Character User Interfaces (CUIs), Graphical User Interfaces (GUIs), Auditory User Interfaces, and Multimodal User Interfaces.

Clearly, ActionGUI is designed for a different purpose than XUL and UsiXML. In particular, ActionGUI is designed for developing *secure* data-management applications. A key design decision for ActionGUI was to ensure that the security model and the GUI models "speak" the same language. To the best of our knowledge, neither XUL nor UsiXML are concerned with security aspects of the UIs. Moreover, ActionGUI is designed for the *model-driven* development of secure data-management applications, and this has two clear consequences. First, ActionGUI's modeling languages are designed to be technology-agnostics, in contrast with XUL, which is tightly linked to Mozilla-related technologies. Second, ActionGUI's modeling languages are designed to support the automatic generation of ready to be deployed applications from the models. As a result, ActionGUI models are more concrete than general UsiXML models, which can be defined at any of the four abstraction levels specified in the Cameleon Reference Framework (CRF) [25]. In particular, a GUI modeler always works at the CRF-Concrete UI level, while the WAR (Web application ARchive) file generated from a GUI model (along with the associated security and data models), belongs to the CRF-Final UI level.

Finally, we note that [83] has carried out promising work on extending our methodology to cope with business processes, which are typically defined at the Task & Concepts abstraction level in CRF. Along these lines, it would be interesting to investigate ways of extending our methodology to support UI modeling at the CRF-Abstract UI level, where interaction details are abstracted away.

Developing secure data-management applications There are also other tools like WebRatio [84], Olivanova [26], and Lightswitch [68] that support development methodolo-

gies for building data-management applications that are similar to the ActionGUI methodology. In these tools, application development starts by building a data model that reflects the data structure required for the database. The development process continues by applying different UI generation patterns to the data model. These patterns enable data retrieval, data editing, data creation, and database search. In contrast to what these tools can provide, the ActionGUI toolkit offers developers the full flexibility to create designs, without burdening them with the restrictions imposed by the obligatory use of a fixed number of given patterns. The above tools also impose a major restriction at the level of data management, i.e., at the level of data access and visualization: the information that can be referenced and therefore that can be accessed and visualized within one screen, can only come from one table of the database or, at most, from two tables that are reachable from each other within one navigation step.

The three tools, WebRatio, Olivanova, and Lightswitch, support the definition and generation of RBAC policies at different granularity levels. Lightswitch supports granting or denying permissions (whose actual behavior must be manually programmed) to execute create, read, update, or delete actions on entities for users in different roles. WebRatio and Olivanova also support granting or denying permissions to execute a similar set of actions on entity's properties, individually, for users in different roles. In WebRatio and Olivanova, the role of the authorization constraints could be played by preconditions restraining the invocation of actions through a concrete UI. Note, however, that none of these tools implement an algorithm capable of lifting to the UIs, the security policy that governs the access to data.

Chapter 6

Concluding remarks and future work

The methodology we propose constitutes a further development of the idea of model-driven security [9]. The two main innovations are an expressive language for modeling an application's graphical user interface and behavior, and a many-models-to-model transformation that lifts a security policy specified on the application's data model to this behavioral model. Our transformation function captures the idea that authorization policies regulating complex transactions can be generated uniformly from much simpler policies on data. Despite our use of expressive modeling languages, we have shown for data-management applications that it is possible to generate automatically complete deployable applications. Our methodology is supported by the ActionGUI toolkit. The reported applications show the toolkit's potential for developing real-world applications.

In the future we plan to further develop the ActionGUI project in two main directions:

Enhancing the ActionGUI toolkit. There is still much work ahead to turn this toolkit into a full, robust, industrial-strength development platform. In the short term, we plan to develop improved model editors and better support for integrating custom code. Then, in the medium term, we would also like to support model analysis, based on the formal semantics of our models and on the correctness of our model transformation. The following are examples of questions we would like to be able to formally answer. Will every sequence of action executed by every event in the model, preserve the data model's invariants? Will authorization checks ever force a transaction roll-back? Do the conditions in the GUI model make redundant the authorization checks, generated by the model transformation? Analysis support would allow us to optimize generated code and support assurance activities like system certification. Finally, in the long term, we would like to support GUIs running on different platforms, like mobile devices. We also plan to add support for handling *privacy policies*: modeling and generating code to enforce that data usage must follow the purpose for which the data was collected and may entail obligations.

Extending the applicability of the ActionGUI methodology and tools. SecureDAO is a Java library developed by Gonzalo Ortiz de Jaureguizar to enforce fine-grained access control within ActionGUI applications. It is designed to execute any of the so-called CRUD actions (i.e, create, read, update, and delete data), but only if the authorization constraint specified for this action in the given security model is satisfied.

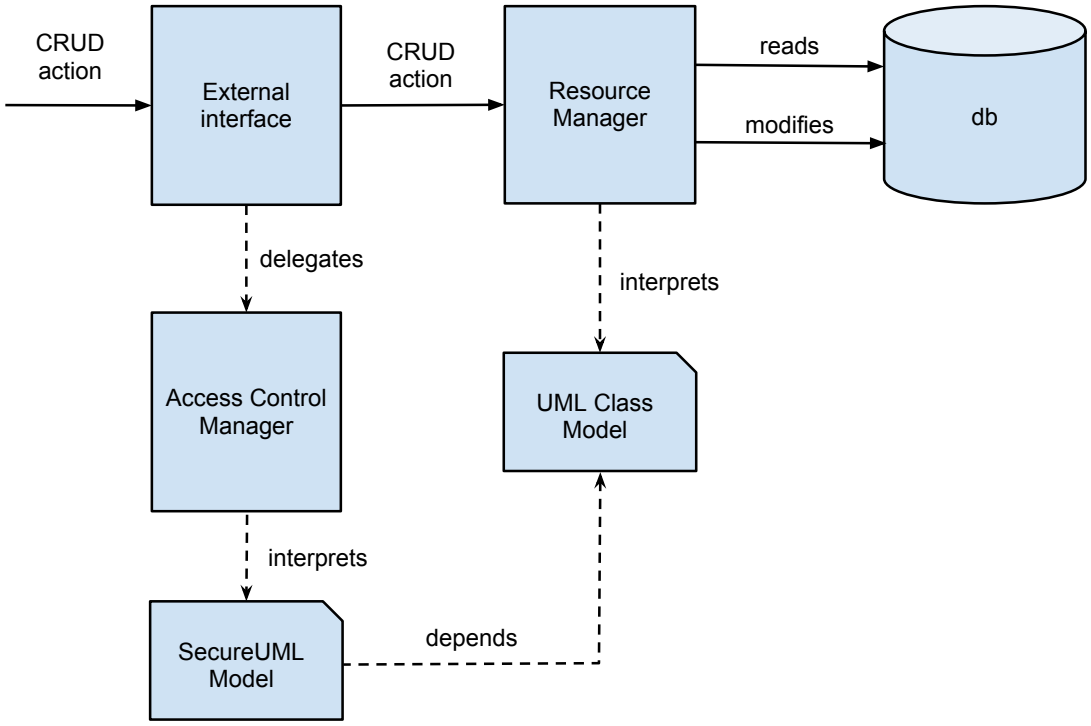


Figure 6.1: SecureDAO's Current Architecture.

The CRUD actions are executed by the SecureDAO's *resource manager*, while the authorization constraints are checked by the SecureDAO's *access control manager*. SecureDAO has a very modular architecture, which is illustrated in Figure 6.1.

In the future, we plan to develop an enterprise-oriented model-driven security enforcement mechanism, by extending SecureDAO in the following dimensions:

- *SecureDAO as a service*, so that not only one but multiple applications could manage the same resources with the same access control policies.
- A *more general resource manager*, so that not only data but also methods could be managed.
- A *more general access control manager*, so that not only SecureUML-based policies but other well-defined security policies could be effectively checked.

To accomplish these goals we will need to address the following research and technological challenges:

- SecureDAO is currently implemented as a Java library. To provide SecureDAO as a service we need to wrap it in an enterprise application that could be used by different customers applications through application level entry points. Our challenge here is to automatically provide type-safe, easy-to-use SecureDAO's entry points, based on the customer application's underlying data model.
- SecureDAO's resource manager's applicability is currently limited to data which is stored in MySQL databases that use very specific databases schemes. A more general resource manager should, first of all, be able to manage data stored in different (also

non-relational) databases. Our challenge here is to build a generic resource manager, which could be instantiated by providing a well-defined mapping from the customer application's underlying data model, to the customer application's database schema. Furthermore, a more general resource manager should be able to execute also non-CRUD actions (i.e., general methods). Our challenge here is to provide a clean connection between the resource manager and the libraries implementing the required non-CRUD actions.

- SecureDAO's access control manager applicability is currently limited to policies which are declared using SecureUML models, and whose authorization constraints are formalized using the Object Constraint Language (OCL) [73]. Our challenge here is to build a generic access control manager, which could be instantiated by providing a well-defined evaluation semantics of an arbitrary authorization language.

Appendices

Appendix A

ComponentUML EBNF syntax

Tokens

data_unit_name ::= [‘A’-‘Z’] ([‘0’-‘9’,‘A’-‘Z’,‘a’-‘z’])* ;

property_name ::= [‘a’-‘z’] ([‘0’-‘9’,‘A’-‘Z’,‘a’-‘z’])* ;

enum_literal_name ::= ([‘A’-‘Z’,‘_’])+ ;

collection_type ::= ‘Set’ | ‘Bag’ | ‘OrderedSet’ | ‘Sequence’ ;

primitive_type ::= ‘Integer’ | ‘Boolean’ | ‘String’ | ‘Real’ ;

eol ::= \n | \r | \r\n ;

Production rules

<DataModel> ::= **eol*** <ListOfDataUnitsDec>? ;

<ListOfDataUnitsDec> ::= <DataUnitDec> **eol*** <ListOfDataUnitsDec>? ;

<DataUnitDec> ::= <EntityDec> | <EnumDec> ;

<EntityDec> ::= ‘entity’ **data_unit_name** (‘extends’ **data_unit_name**)?
‘{’ **eol**+ <EntityBody>? ‘}’ ;

<EntityBody> ::= <PropertyDec> **eol**+ <EntityBody>? ;

<PropertyDec> ::= <PropertyTypeDec> **property_name**
(‘oppositeTo’ **property_name**)? ;

<PropertyTypeDec> ::= <CollectionTypeDec> | <BasicTypeDec> ;

<CollectionTypeDec> ::= **collection_type** ‘(’ <PropertyTypeDec> ‘)’ ;

<BasicTypeDec> ::= **primitive_type** | **data_unit_name** ;

<EnumDec> ::= 'enum' **data_unit_name** '{' **eol**+ <EnumBody> '}' ;

<EnumBody> ::= **enum_literal_name** **eol**+ <EnumBody>? ;

Appendix B

SecureUML EBNF syntax

Tokens

identifier ::= (['0'-'9','A'-'Z','a'-'z','_'])+ ;

action_type ::= 'fullAccess' | 'create' | 'delete' | 'read' |
'update' | 'add' | 'remove' ;

eol ::= \n | \r | \r\n ;

ocl_expr ::= '[' (-[ ])+ ']' ;

Production rules

<SecurityModel> ::= **eol*** <ListOfRolesDec>? ;

<ListOfRolesDec> ::= <RoleDec> **eol*** <ListOfRolesDec>? ;

<RoleDec> ::= 'role' **identifer** ('extends' **identifer**)? **eol***
'{' **eol*** <RoleBody>? '}' ;

<RoleBody> ::= <ListOfPermissionsDec> ;

<ListOfPermissionsDec> ::= <PermissionDec> **eol*** <ListOfPermissionsDec>? ;

<PermissionDec> ::= **identifer eol*** '{' **eol*** <PermissionBody>? '}' ;

<PermissionBody> ::= <ListOfListsOfActions> ;

<ListOfListsOfActions> ::= <ListOfActions> **eol+** <ListOfListsOfActions>? ;

<ListOfActions> ::= <ActionDec> <RestListOfActions>? <ConstraintDec>? ;

<ConstraintDec> ::= 'constrainedBy' **eol*** **ocl_expr** ;

<ActionDec> ::= **action_type** <ResourceActionDec>? ;

<ResourceActionDec> ::= '(' **identifer** ')';

<RestListOfActions> ::= ',' **eol*** <ActionDec> <RestListOfActions>? ;

Appendix C

GUIML EBNF syntax

Tokens

identifer ::= [‘A’-‘Z’,‘a’-‘z’,‘_’] ([‘0’-‘9’,‘A’-‘Z’,‘a’-‘z’,‘_’])*;

simple_widget_type ::= ‘Label’ | ‘Button’ | ‘TextField’ |
‘PasswordField’ | ‘BooleanField’ | ‘DateField’ ;

primitive_type ::= ‘Integer’ | ‘Boolean’ | ‘String’ | ‘Real’ ;

collection_type ::= ‘Set’ | ‘Bag’ | ‘OrderedSet’ | ‘Sequence’ ;

ocl_expr_part ::= (-[‘[’,‘]’,‘$’])+;

exception_type ::= ‘SecurityException’ | ‘ModelException’ | ‘Exception’ ;

Production rules

<GUIModel> ::= <ListOfWidgetsDec>? ;

<Path> ::= **identifier** (‘.’ **identifier**)* ;

<ListOfWidgetsDec> ::= <WidgetDec> <ListOfWidgetsDec>? ;

<WidgetDec> ::= <WindowDec> | <TableDec> | <ComboBoxDec> | <SimpleDec> ;

<WindowDec> ::= ‘Window’ <Path> ‘isMain’? ‘{’ <ListOfVarsDec>?
<ListOfWidgetsDec>? <ListOfEventsDec>? ‘}’ ;

<TableDec> ::= ‘Table’ <Path> ‘{’ <ListOfVarsDec>? <ColumnBlock>
<ListOfEventsDec>? ‘}’ ;

<ComboBoxDec> ::= ‘ComboBox’ <Path> ‘{’ <ListOfVarsDec>? <WidgetDec>
<ListOfEventsDec>? ‘}’ ;

<SimpleDec> ::= **simple_widget_type** <Path> '{' <ListOfVarsDec>?
<ListOfEventsDec>? '}' ;

<ColumnBlock> ::= 'columns' '{' <ListOfColumnsDec> '}' ;

<ListOfColumnsDec> ::= <GUIExpr> ':' <WidgetDec> <ListOfColumnsDec>? ;

<ListOfVarsDec> ::= <VarDec> <ListOfVarsDec>? ;

<VarDec> ::= TypeDec **identifier** (':=' <GUIExpr>)?
('{' <ListOfActionsDec> '}')? ;

<TypeDec> ::= <PrimitiveType> | <EntityType> |
<CollectionType> | <TupleType> ;

<PrimitiveType> ::= **primitive_type** ;

<EntityType> ::= **identifier** ;

<CollectionType> ::= **collection_type** '(' <TypeDec> ')' ;

<TupleType> ::= 'Tuple' '(' <ListOfTupleAttrs> ')' ;

<ListOfTupleAttrs> ::= **identifier** ':' <TypeDec> (',' <ListOfTupleAttrs>)? ;

<ListOfEventsDec> ::= <EventDec> <ListOfEventsDec>? ;

<EventDec> ::= <OnCreateDec> | <OnClickDec> | <OnViewDec>

<OnCreateDec> ::= 'event' 'onCreate' '{' <ListOfActionsDec>? '}';

<OnClickDec> ::= 'event' 'onClick' '{' <ListOfActionsDec>? '}';

<OnViewDec> ::= 'event' 'onView' '(' **identifier** ')'
'{' <ListOfActionsDec>? '}';

<ListOfActionsDec> ::= <ActionDec> <ListOfActionsDec>? ;

<ActionDec> ::= <Path> ':=' 'new' **identifier**
| <Path> ':=' <GUIExpr>
| <Path> '+=' <GUIExpr>
| <Path> '-=' <GUIExpr>
| 'delete' <GUIExpr>
| <GUIExpr> ':=' <GUIExpr>
| <GUIExpr> '+=' <GUIExpr>
| <GUIExpr> '-=' <GUIExpr>
| 'reevaluate' <Path>
| 'open' **identifier** ('(' <ListOfOpenArgsDec> ')')*
| 'back'

| **identifier** ‘(’ <ListOfOperationArgsDec>? ‘)’
| ‘if’ <GUIExpr> ‘{’ <ListOfActionsDec>? ‘}’
| ‘if’ <GUIExpr> ‘{’ <ListOfActionsDec>? ‘}’ ‘else’
‘{’ <ListOfActionsDec>? ‘}’
| ‘foreach’ **identifier** ‘in’ <GUIExpr> ‘{’ <ListOfActionsDec>? ‘}’
| ‘notification’ ‘(’ <GUIExpr> ‘,’ <GUIExpr> ‘,’ <GUIExpr>
| ‘try’ ‘{’ <ListOfActionsDec> ‘}’ <CatchDec>+
| ‘exit’;

<ListOfOpenArgsDec> ::= <OpenArgDec> (‘,’ <ListOfOpenArgsDec>)? ;

<OpenArgDec> ::= **identifier** ‘:’ <GUIExpr> ;

<ListOfOperationArgsDec> ::= <GUIExpr> (‘,’ <ListOfOperationArgsDec>)? ;

<CatchDec> ::= ‘catch’ ‘(’ <CaughtExceptionsDec> **identifier**? ‘)’
‘{’ <ListOfActionsDec>? ‘}’;

<CaughtExceptionsDec> ::= **exception_type** |
exception_type ‘or’ <CaughtExceptionsDec>;

<GUIExpr> ::= ‘[’ <ListOfGUIExprParts> ‘]‘ ;

<ListOfGUIExprParts> ::= <GUIExprPart> <ListOfGUIExprParts>? ;

<GUIExprPart> ::= **ocl_expr_part** | ‘$’ <Path> ‘$’ ;

Part II

Resumen de la investigación

Capítulo 7

Introducción

La construcción de modelos es la parte principal del diseño de sistemas. Esto es cierto en varias disciplinas de la ingeniería, sobre todo en el caso de la ingeniería del software. En el pasado, los defensores de la Ingeniería Dirigida por Modelos (MDE, de sus siglas en inglés) han sido culpables de pretensiones demasiado ambiciosas: posicionándola como el Santo Grial de la ingeniería del software donde el modelado reemplaza completamente a la programación. En general, esto no es posible, sin embargo, hay dominios especializados donde MDE puede ciertamente desarrollar todo su potencial: en nuestra opinión, las aplicaciones de gestión segura de datos a partir de interfaces gráficas de usuario es uno de ellos.

7.1 Ingeniería del software dirigida por modelos

El continuo aumento del desarrollo y uso de las tecnologías de la información y la comunicación es una fuente constante de problemas de seguridad y fiabilidad. Claramente necesitamos mejores formas de desarrollar sistemas software. La Ingeniería Dirigida por Modelos (MDE, de sus siglas en inglés) [61] es una metodología de desarrollo de software centrada en crear modelos de diferentes vistas del sistema desde los cuales los distintos artefactos del sistema, tales como el código fuente y los datos de configuración, son derivados o, si es posible, generados de manera automática.

Lenguajes de modelado de dominio específico En nuestra opinión, sólo limitando el dominio es posible definir de manera suficientemente precisa lenguajes de modelado que soporten la generación automática de aplicaciones totalmente funcionales. Se puede decir que la tardía adopción de la MDE se debe a la dificultad que conlleva definir lenguajes de modelado de dominio específico que sean efectivos, así como también al esfuerzo requerido por los desarrolladores para aprender lenguajes de modelado y el arte de la construcción de modelos. Definir un buen lenguaje de modelado de dominio específico requiere la búsqueda de las correctas abstracciones y grado de precisión, para capturar los aspectos relevantes de la estructura y la lógica de un sistema software. Además, para que un lenguaje de modelado sea realmente útil y utilizable para los desarrolladores de software se deben proporcionar las herramientas apropiadas para construir modelos, analizarlos y mantenerlos sincronizados con los productos finales.

Transformaciones de modelos En MDE, la manera de usar modelos para producir otros artefactos de desarrollo es mediante la transformación de modelos. Los siguientes tipos de transformaciones has sido ampliamente usadas hasta ahora:

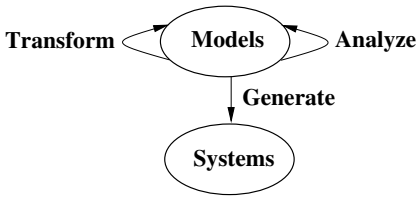


Figura 7.1: Uso de Modelos en Seguridad Dirigida por Modelos

- *Generación de código y ejecución de artefactos:* Los modelos pueden ser mapeados a código u otros artefactos que afectan al comportamiento en tiempo de ejecución de los sistemas. Cuando se genera código, la función de transformación equivale a un tipo de traductor o compilador. Ejemplos de otros artefactos que pueden ser generados a partir de modelos son artefactos de desarrollo o de datos de configuración, los cuales también afectan al comportamiento del sistema.

- *Generación de modelos:* Los modelos pueden ser mapeados a otros modelos. Normalmente dichas transformaciones añaden detalles, especializan construcciones o modifican representaciones. Un ejemplo es la especialización de modelos independientes de plataforma a modelos de plataformas específicas. En general, las transformaciones de modelos soportan el problema de descomposición durante el desarrollo, donde aspectos del diseño pueden ser separados en diferentes modelos, los cuales se componen más adelante.

- *Generación de casos de prueba:* Estos casos pueden generarse desde los modelos.

7.2 Seguridad dirigida por modelos

La seguridad dirigida por modelos (MDS, de sus siglas en inglés) [66, 11, 7, 8, 36, 9, 44] es una especialización de MDE en el dominio de la seguridad. Tal y como se trata en [9], los modelos pueden ser usados para las siguientes cuatro actividades en el desarrollo de sistemas seguros:

A1. Documentación precisa de requisitos de seguridad, en conjunto con requisitos de diseño.

A2. Análisis de requisitos de seguridad.

A3. Transformaciones basadas en modelos, tales como migrar políticas de seguridad de los datos de la aplicación a políticas para otros artefactos o capas del sistema.

A4. Generación de código, incluyendo infraestructuras de seguridad configuradas y completas.

La Figura 7.1 describe estas actividades así como la relación entre ellas. Los diseñadores especifican modelos de diseño de seguridad los cuales combinan seguridad y requisitos de diseño (A1). Cuando los lenguajes de modelado tienen una semántica bien definida, estos diseños pueden analizarse formalmente (A2). En el diseño de sistemas seguros, la seguridad puede ser relevante en diferentes capas o niveles. Mediante el uso de transformaciones de modelos se pueden migrar políticas de seguridad de un modelo a otros modelos (A3). Finalmente, se pueden usar herramientas para generar de manera automática código u otros artefactos del sistema directamente de los modelos (A4).

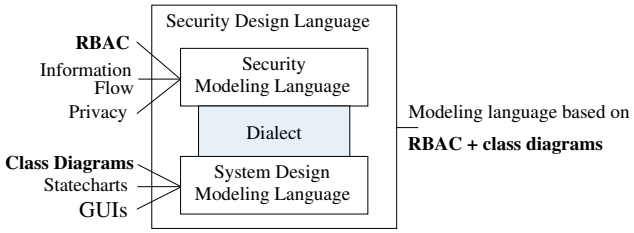


Figura 7.2: Lenguajes de Modelado y su Combinación

La parte crucial de la especialización que la seguridad dirigida por modelos trae consigo, afecta al lenguaje de modelado. En vez de adoptar una estrategia basada en un único lenguaje que sirva para todo, [11] propone un esquema general para integrar requisitos de seguridad en los modelos de diseño del sistema. La idea principal es definir lenguajes de modelado de seguridad genéricos de manera que dejen abierta la naturaleza de los recursos protegidos, es decir, si estos recursos son datos, objetos de negocio, procesos, estados del controlador, etc. La Figura 7.2 muestra ejemplos de diferentes nociones de seguridad las cuales pueden ser especificadas usando un lenguaje de modelado de seguridad (arriba a la izquierda) que puede ser integrado en diferentes lenguajes de modelado de diseño (abajo a la izquierda), dando lugar a un lenguaje de modelado de diseño con seguridad (a la derecha). Por ejemplo, se podría combinar un lenguaje de modelado para Control de Acceso Basado en Roles (RBAC, de sus siglas en inglés) con Diagramas de Clases, tal y como se indica en negrita en la figura. Esta combinación está hecha a partir de la definición de un dialecto que identifica los elementos del lenguaje de diseño como los recursos protegidos del lenguaje de seguridad. De esta manera, se pueden definir lenguajes de manera flexible para formular distintos tipos de diseño de sistemas, junto con sus requisitos de seguridad.

7.3 Aplicaciones de gestión segura de datos

Las aplicaciones de gestión de datos se basan en las conocidas acciones CRUD[1] de crear (create), leer (read), modificar (update) y borrar (delete) datos del almacenamiento persistente. Estas operaciones son los bloques básicos de numerosas aplicaciones, por ejemplo, páginas web dinámicas donde los usuarios crean cuentas, almacenan y modifican información y reciben vistas personalizadas basadas en sus datos almacenados. Cuando los datos gestionados son sensibles la seguridad es una preocupación, por lo que el uso de estas acciones tiene que ser controlado.

El control de acceso es la estrategia estándar para restringir las acciones de los usuarios sobre los datos de la aplicación. Cuando las políticas de control de acceso son suficientemente simples, es posible formalizarlas de una manera declarativa independientemente de la lógica de negocio de la aplicación. Por ejemplo, los sistemas multicapa para aplicaciones web a menudo construyen soporte para el control de acceso basado en roles en el servidor de aplicaciones, el cual es configurado independientemente de los detalles procedimentales de la aplicación. Por el contrario, las políticas de control de acceso de grano fino pueden depender no sólo de las credenciales de usuario sino también de la satisfacibilidad de restricciones en el estado de la capa persistente, es decir, en los valores de los elementos almacenados. En tales casos, las comprobaciones de autorización típicamente son implementadas mediante programación, codificándolas directamente en los lugares apropiados

[1]Esto es cierto para aplicaciones cuya capa persistente es una base de datos. Cuando es un LDAP o un almacenamiento en la nube, no son exactamente las mismas acciones.

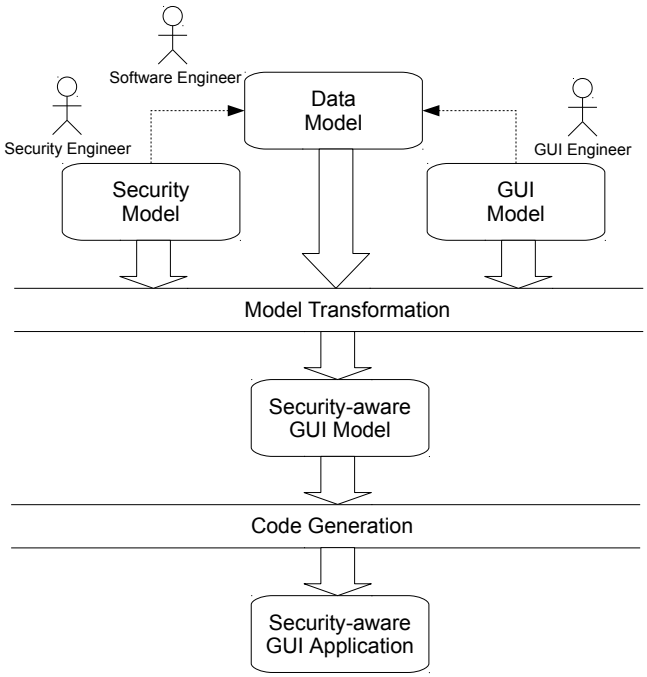


Figura 7.3: Desarrollo dirigido por modelos de GUIs con seguridad.

del código de la aplicación. Desafortunadamente, estas codificaciones son difíciles de manejar, propensas a errores y escalan de forma pésima. Además, son difíciles de auditar y mantener ya que las comprobaciones de autorización están esparcidas a lo largo del código, por lo que cambios en la política de seguridad requieren cambios en el código.

7.4 Desarrollo dirigido por modelos de aplicaciones de gestión segura de datos

En este trabajo proponemos una metodología de desarrollo dirigido por modelos para aplicaciones seguras de gestión de datos. Consiste en un conjunto de lenguajes para modelar sistemas multicapa y en un conjunto de herramientas para generar estas aplicaciones. En nuestra metodología, una aplicación de gestión segura de datos es modelada a partir de tres modelos relacionados entre ellos:

1. Un *modelo de datos* define la estructura de datos de la aplicación en términos de sus clases, atributos, asociaciones y métodos (que no modifican el estado);

2. Un *modelo de seguridad* define la política de seguridad de la aplicación en términos de acceso autorizado a las acciones sobre los recursos proporcionados por el modelo de datos.

3. Una interfaz gráfica de usuario o *modelo GUI* define la interfaz gráfica de la aplicación y la lógica de la aplicación. En particular este modelo formaliza ambos, la *estructura de la GUI* y el *comportamiento*.

El núcleo de esta metodología, ilustrado en la Figura 7.3, es una función de transformación de modelos que automáticamente traslada la política que está especificada en el modelo de seguridad al modelo de GUI. La idea es simple pero poderosa. El modelo

de seguridad especifica bajo qué condiciones las acciones sobre datos están autorizadas. La información de control en el modelo de GUI especifica qué acciones son ejecutadas en respuesta a qué eventos. Trasladar la política esencialmente consiste en prefijar cada una de las acciones sobre datos en el modelo de GUI con la comprobación de autorización especificada en el modelo de seguridad. El modelo de GUI resultante contiene la especificación de seguridad. En concreto, especifica la estructura de la GUI, el flujo de información con el almacenamiento persistente y todas las comprobaciones de autorización.

Hemos implementado esta metodología con un conjunto de herramientas, llamado ActionGUI [1], el cual implementa esta transformación de modelos. A partir del modelo de GUI con seguridad resultante, ActionGUI genera una aplicación desplegable junto con todo el soporte para el control de acceso, basada en la siguiente arquitectura estándar de tres capas.

1. Capa de presentación (también conocida como front-end): Los usuarios acceden a las aplicaciones web a través de navegadores estándares, los cuales renderizan el contenido (HTML y JavaScript) proporcionado por el servidor web de forma dinámica.

2. Capa de aplicación: El conjunto de herramientas genera Aplicaciones Web Java. Las aplicaciones corren en un servidor web (como Tomcat o GlassFish), procesan las solicitudes de los clientes y generan contenido el cual es enviado de vuelta al cliente para ser renderizado. Cuando las solicitudes de los clientes son procesadas la aplicaciones generadas *interpretan* su modelo de GUI con seguridad subyacente.

3. Capa persistente: La aplicación generada gestiona la información almacenada en una base de datos. Para cada aplicación, el conjunto de herramientas genera el correspondiente esquema de la base de datos a partir del modelo de datos de la aplicación.

7.4.1 Las principales contribuciones del autor

ActionGUI es el último resultado de un proyecto muy ambicioso en el que numerosos investigadores han contribuido de diferentes maneras durante los últimos siete años. Los comienzos de este proyecto se remontan a 2008 cuando Manuel Clavel, Viviane da Silva, Christiano Braga y Marina Egea llevaron a cabo el proyecto reportado en [36]. El elemento principal tras este trascendental trabajo fue el uso de la seguridad dirigida por modelos para desarrollar una aplicación industrial. Los autores tuvieron éxito en modelar, usando SecureUML [11], la política de grano fino de control de acceso de la aplicación. Sin embargo, el proceso de codificar manualmente la política modelada en la interfaz gráfica de usuario de la aplicación — liderada por Felipe Padilha, un estudiante de Christiano Braga— consumió demasiado tiempo y fue propenso a errores. Este resultado insatisfactorio proporcionó la *raison d'être* para el proyecto ActionGUI: la búsqueda de una solución dirigida por modelos al problema de trasladar la política de control de acceso, típicamente especificada sobre los datos de la aplicación, a la interfaz gráfica de usuario de la aplicación. Además, proporcionó los *criterios de éxito* del proyecto: a saber, que la solución prevista debe soportar un desarrollo dirigido por modelos (de modelos a código) y debe ser capaz de tratar aplicaciones de tamaño industrial.

Como se detallará más abajo, las contribuciones del autor al proyecto ActionGUI han estado siempre dirigidas hacia la satisfacción de los anteriormente mencionados criterios de éxito. De hecho, se puede decir con total sinceridad que él es el responsable del grado al que estos criterios de éxito son satisfechos, gracias a su ininterrumpido trabajo en las

sucesivas versiones del conjunto de herramientas de ActionGUI y sus reiterados esfuerzos en aplicar la metodología ActionGUI en casos de estudio de complejidad cada vez mayor.

La primera solución concreta dirigida por modelos para nuestro problema llegó en 2009. En particular, Manuel Clavel en colaboración con David Basin (ETH), Marina Egea y su estudiante Michael Schläpfer introdujeron por primera vez en [77, 13] la idea de usar una transformación de muchos modelos a un modelo para construir un modelo de GUI *seguro* con respecto a un modelo de seguridad. La primera implementación de esta transformación la hizo Michael, la cual se explica en [76]. Esta solución fue pronto soportada por SSG [44], un novedoso entorno de desarrollo para construir GUIs seguras automáticamente. SSG fue la primera contribución del autor al proyecto. Consistía en una serie de plugins de Eclipse que incluían tres editores de modelos, una herramienta de transformación de modelos y un generador de código. Una parte clave en el trascendental generador de código de SSG fue EOS [33], un componente Java diseñado e implementado por el autor para evaluar de manera eficiente expresiones OCL que aparecen tanto en los modelos de seguridad (restricciones de autorización), como en los modelos de GUI (condiciones y argumentos de acciones). EOS fue posteriormente reemplazado por MySQL4OCL [50], implementado por Carolina Dania bajo la dirección de Marina Egea. El mapeo de expresiones OCL a procedimientos almacenados MySQL con MySQL4OCL tenía la ventaja sobre EOS de evaluar expresiones OCL directamente sobre la base da datos de la aplicación. Finalmente, MySQL4OCL fue a su vez reemplazado por un nuevo evaluador OCL, implementado por Gonzalo Ortiz Jaureguizar, el cual integraba de manera satisfactoria una solución híbrida con las mejores características de EOS y MySQL4OCL. Aunque no inicialmente integrado en SSG, el autor también implementó durante ese tiempo un mapeo de OCL a lógica de primer orden para la comprobación de restricciones OCL [34] usando resolutores de SMT. Este mapeo fue revisado y extendido para los valores OclUndefined y OclInvalid en [39] por Carolina Dania y Manuel Clavel, el cual está actualmente integrado en el conjunto de herramientas de ActionGUI, en parte gracias al autor. Entre otras aplicaciones, este mapeo ha sido usado de manera satisfactoria para razonar formalmente sobre políticas de control de acceso de grano fino [43].

De acuerdo con los criterios de éxito del proyecto, era necesario probar la solución dirigida por modelos y el conjunto de herramientas propuestos en un escenario real. Para este fin, el autor obtuvo una beca de un año de la Fundación Vodafone (Septiembre de 2010 – Agosto de 2011) para desarrollar una solución CRM y un sistema de gestión de voluntarios para un hospital y un centro de cuidados paliativos de Madrid. Para llevar a cabo estas aplicaciones, las cuales fueron finalmente reportadas en [12], el autor tuvo que extender de varias maneras decisivas la solución originalmente propuesta en [13], además del conjunto de herramientas asociado [44]. En particular, a nivel de lenguaje de modelado de GUIs el autor introdujo *variables de widget* como un elemento de modelado clave. Las variables de widget son variables que poseen los widgets. En principio, son declaradas por el modelador pero también hay variables de widget que, por defecto, posee cada widget según su tipo. Los valores de algunas variables de widget son manejados de formas especiales. Por ejemplo, las variables caller y role pertenecen por defecto a cada ventana. Ellas almacenan, respectivamente, el usuario de la aplicación y el rol del usuario y sus valores se pasan de forma implícita como argumentos en cada acción de abrir una nueva ventana. Además de las variables de widget, el autor introdujo *tablas* y *listas desplegables* en el lenguaje de GUI como nuevos tipos de widgets con sus respectivas variables de widget por defecto: rows para almacenar la colección de elementos que pueden ser seleccionados, row para referenciar a cada uno de los elementos de la colección y selected para referenciar a los elementos seleccionados (si existen) por el usuario. Luego, a nivel de lenguaje de modelado

de políticas de seguridad el autor extendió SecureUML en dos formas cruciales. Primero, reemplazando la acción de modificar (update) un enlace entre dos objetos (cuyo significado era originalmente ambiguo debido a que podía implicar crear un enlace, borrar un enlace o ambas) por dos nuevas acciones separadas, una para *crear* un enlace y otra para *borrar* un enlace. Segundo, el autor extendió SecureUML permitiendo dos nuevas variables especiales, value y target, en las restricciones de autorización: la primera refiriéndose al valor que será usado para modificar un atributo, si el permiso es concedido. La segunda refiriéndose al objeto que será enlazado (o desenlazado) en el otro extremo de la asociación, si el permiso es concedido. Lógicamente, estos cambios en los lenguajes de modelado tuvieron un impacto directo y de gran tamaño en el conjunto de herramientas que soportan la metodología. El autor, como principal responsable en ese momento del conjunto de herramientas de ActionGUI, fue el que estuvo al cargo de implementar, de una manera efectiva, todas las nuevas características de modelado anteriormente mencionadas. La solución mejorada para nuestro problema, junto con el conjunto de herramientas asociadas, fue por primera vez presentada en [10].

El siguiente mayor caso de estudio fue llevado a cabo por el autor en el contexto del proyecto NESSoS (Octubre de 2010 – Marzo de 2014). NESSoS (de sus siglas en inglés) es la Red de Excelencia sobre la creación de Servicios y Sistemas de Software Seguros para el Internet del Futuro [71]. El caso de estudio, el cual fue reportado en [42], consiste en un sistema web para la gestión de registros electrónicos de salud (EHRM, de sus siglas en inglés). Los registros electrónicos de salud (EHR, de sus siglas en inglés) almacenan información creada por, o en nombre de, un profesional de la salud en el contexto del cuidado de un paciente. La especificación de la aplicación EHRM de NESSoS define diferentes casos de uso del sistema junto con la política de control de acceso asociada. Los casos de uso incluyen: registrar nuevos pacientes en un hospital y asignarles profesionales de la salud tales como enfermeros o doctores, recuperar la información de un paciente, registrar nuevos enfermeros y doctores y asignarlos a un departamento, cambiar enfermeros o doctores de un departamento a otro y reasignar pacientes a doctores. Mientras se llevó a cabo el caso de estudio de NESSoS, el autor descubrió una gran limitación en la transformación de modelos para trasladar la política de seguridad a nivel de GUI. En el trabajo previo [13, 10], esta traslación consistió en prefijar cada evento del modelo de GUI con la comprobación de autorización especificada en el modelo de seguridad. Sin embargo, el autor se dio cuenta de que, debido a que las acciones sobre datos que se ejecutan en un evento pueden cambiar el estado de la capa persistente, comprobar la autorización a nivel de eventos y por lo tanto antes de ejecutar cada una de las acciones sobre datos, era suficiente sólo si la política de seguridad subyacente no contenía restricciones de autorización (las cuales son asumidas de manera explícita en [13]) o si las contenía, no dependían de los valores que eran modificados durante la ejecución de las acciones sobre datos del evento (tal y como ocurre en los ejemplos tratados en [10]). Para resolver esta limitación, el autor propuso comprobar las autorizaciones entes de ejecutar cada una de las acciones sobre datos del evento dotando a los eventos de una semántica de *transacción*: o todas las acciones sobre datos son ejecutadas en el orden dado o ninguna de ellas es ejecutada. Esta importante generalización de [13, 10] apareció por primera vez en [12, 6] y está actualmente soportada en el conjunto de herramientas de ActionGUI.

Por último, si bien no menos importante, el autor ha contribuido de manera significativa al proyecto ActionGUI escribiendo el manual de usuario y el manual de instalación del conjunto de herramientas de ActionGUI, así como la documentación técnica de los lenguajes de modelado de ActionGUI. Es meritorio mencionar que este material (en sus diferentes versiones) ha sido usado de manera satisfactoria por estudiantes e instructores

en varios cursos, incluyendo: "Generación Automática de Modelos de GUI Inteligentes y Seguros." (Seminario de Ingeniería Dirigida por Modelos, IRISA, Rennes, Francia, Noviembre de 2009); "Seguridad dirigida por modelos: fundamentos, herramientas y práctica" (11 Escuela Internacional de Fundamentos de Análisis y Diseño de Seguridad, Bertinoro, Italia, Septiembre de 2011); "Ingeniería de Seguridad" (curso de nivel de máster, ETH Zúrich, Suiza, Octubre de 2012 – Enero de 2013); "ActionGUI Day" (Día de formación industrial para el uso de ActionGUI, IMDEA Software, Junio de 2013) e "Ingeniería Dirigida por Modelos en Acción" (curso de facultad, Universidad Industrial de la Ciudad de Ho Chi Minh, Vietnam, Enero de 2014 – Febrero de 2014). Las últimas versiones de los manuales de usuario y de instalación de ActionGUI, están disponibles en la página web de ActionGUI [1].

7.5 Resumen

- Capítulo 8: *Una metodología para el desarrollo de aplicaciones de gestión segura de datos.* Presentamos ActionGUI como una metodología innovadora dirigida por modelos para el desarrollo de aplicaciones de gestión segura de datos. Los desarrolladores de sistemas proceden modelando las tres vistas diferentes de la aplicación deseada: el modelo de datos, el modelo de seguridad y el modelo de GUI. Estos modelos formalizan respectivamente el dominio de los datos de la aplicación, la política de autorización y la interfaz gráfica junto con el comportamiento de la aplicación. Después, una función de transformación de modelos traslada la política especificada por el modelo de seguridad al modelo de GUI. Esto permite una separación de tareas, donde el comportamiento y la seguridad son especificados de manera separada y posteriormente son combinados para generar un modelo GUI con seguridad.

- Capítulo 9: *Soporte para nuestra metodología de desarrollo de aplicaciones de gestión segura de datos.* Tras un breve informe del estado actual del conjunto de herramientas de ActionGUI, reportamos nuestra experiencia en el desarrollo del evaluador Eye OCL Software (EOS), un componente Java para la evaluación eficiente de OCL en escenarios mediano-grandes y también exploramos varias estrategias de evaluación de expresiones OCL en escenarios realmente grandes. A continuación, proponemos un mapeo de un subconjunto de OCL a lógica de primer orden (FOL) y usamos este mapeo para comprobar la insatisfacibilidad de conjuntos de restricciones OCL. Argumentamos que nuestro mapeo es simple, ya que las sentencias FOL resultantes son muy similares a las restricciones OCL originales, y también práctico, ya que podemos usar herramientas de razonamiento automático, tales como demostradores automáticos de teoremas y resolutores SMT, para comprobar automáticamente la insatisfacibilidad de conjuntos no triviales de restricciones OCL. Finalmente, proponemos una metodología para el razonamiento sobre políticas de control de acceso de grano fino, cuyas restricciones de autorización son especificadas en OCL mediante el uso del mapeo de OCL a FOL anteriormente mencionado.

- Capítulo 10: *Desarrollo de aplicaciones de gestión segura de datos con nuestra metodología.* Proporcionamos un informe detallado de nuestro uso de ActionGUI para desarrollar una aplicación de gestión segura de datos. Esta aplicación está basada en un caso de estudio propuesto en NESSoS (de sus siglas en inglés), la Red de Excelencia sobre la creación de Servicios y Sistemas de Software Seguros para el Internet del Futuro [71]. El caso de estudio eHealth consiste en un sistema web para la gestión de registros electrónicos de salud (EHRM, de sus siglas en inglés). Los

registros electrónicos de salud almacenan información creada por, o en nombre de, un profesional de la salud en el contexto del cuidado de un paciente. El caso de estudio eHealth es interesante como ejemplo de desarrollo de una aplicación de gestión segura de datos, proporcionando una prueba de concepto de la aplicación de la metodología ActionGUI sobre un problema relevante a nivel industrial. Para proporcionar más evidencias de la usabilidad de nuestra tecnología, incluímos en este Capítulo un resumen de otras cuatro aplicaciones web que han sido desarrolladas usando ActionGUI.

- Capítulo 11: *Trabajo relacionado*. Primero nos centramos en los trabajos que están directamente relacionados con nuestra principal contribución, a saber, ActionGUI: una novedosa metodología dirigida por modelos y con soporte de herramienta para el desarrollo de aplicaciones de gestión segura de datos. Segundo, comparamos ActionGUI con otras herramientas (en este caso herramientas comerciales) para el desarrollo de aplicaciones de gestión segura de datos.

- Capítulo 12: *Conclusiones y trabajo futuro*. Discutimos las dos principales líneas en las cuales se continuará el desarrollo del proyecto ActionGUI. Primero, planeamos mejorar el conjunto de herramientas ActionGUI para convertirlo en una robusta y total plataforma de desarrollo a nivel industrial. Segundo, planeamos extender la aplicabilidad de la tecnología ActionGUI a partir de la generalización de sus principales componentes tecnológicos, implementándolos como servicios software (SaaS, de sus siglas en inglés).

Capítulo 8

Modelado de aplicaciones seguras de gestión de datos

En este Capítulo presentamos ActionGUI como una novedosa metodología para el desarrollo de aplicaciones de gestión segura de datos. Los desarrolladores de sistemas proceden modelando tres vistas diferentes de la aplicación deseada: el modelo de datos, el modelo de seguridad y el modelo de GUI. Estos modelos formalizan respectivamente el dominio de datos de la aplicación, la política de autorización y la interfaz de usuario junto con el comportamiento de la aplicación. Después, una función de transformación de modelos traslada la política especificada en el modelo de seguridad al modelo de GUI. Esto permite una separación de tareas, donde el comportamiento y la seguridad de la aplicación son especificados de forma separada y posteriormente combinada para generar un modelo de GUI con seguridad.

En [10, 13, 44] propusimos la idea de usar transformaciones de modelos para elevar la política de seguridad formulada en términos del modelo de datos al modelo de GUI. Aquí mejoramos y generalizamos este trabajo previo. Trasladar consistía previamente en prefijar cada evento en el modelo de GUI con la restricción de autorización especificada en el modelo de seguridad. Sin embargo, debido a que las acciones sobre datos ejecutadas por un evento pueden cambiar el estado de la capa persistente, el chequeo de autorización a nivel de los eventos, y por lo tanto antes de ejecutar ninguna acción, es suficiente sólo si la política de seguridad subyacente no contiene restricciones de autorización (lo cual se asumía de manera implícita en [13]), o no depende de valores que han cambiado durante la ejecución de acciones sobre datos de un evento (como era el caso en los ejemplos analizados en [10]). Para vencer esta limitación ahora comprobamos las autorizaciones antes de ejecutar cada una de las acciones sobre datos de un evento, proporcionando a los eventos una semántica de *transacción*: o todas las acciones sobre datos son ejecutadas en el orden dado o ninguna de ellas es ejecutada. La descripción formal completa de nuestra metodología se detalla en el informe técnico [6]. En su lugar, aquí proporcionamos una descripción a alto nivel de la corrección de la función de transformación de modelos sobre la cual radica el núcleo de nuestra metodología.

8.1 Background

Para modelar la estructura de datos y la política de seguridad de una aplicación hacemos uso de los ya existentes lenguajes de modelado llamados ComponentUML y SecureUML [11]. En esta Sección introducimos de manera breve estos lenguajes. Ya que SecureUML usa el Lenguaje de Restricción de Objetos (OCL, de sus siglas en inglés) [73]

para modelar restricciones de autorización, también resumiremos sus principales características.

8.1.1 ComponentUML

Los modelos de datos proporcionan una visión orientada de los datos de un sistema. Típicamente son usados para especificar cómo los datos son estructurados, el formato de cada uno de los elementos que lo compone y su organización lógica, es decir, cómo se agrupan y se relacionan los elementos que componen el modelo de datos. Nuestra metodología usa ComponentUML para el modelado de datos. ComponentUML proporciona un subconjunto de los diagramas de clases UML donde las *entidades* (clases) pueden estar relacionadas mediante *asociaciones* y pueden tener *atributos* y *métodos*. En ComponentUML las asociaciones son binarias: están compuestas siempre de dos *extremos de asociación* que conectan dos (no necesariamente distintas) entidades. En el Apéndice A se encuentra la definición completa de la sintaxis de ComponentUML.

8.1.2 Lenguaje de Restricción de Objetos (OCL)

El Lenguaje de Restricción de Objetos (OCL, de sus siglas en inglés) [73] es un lenguaje para especificar restricciones y consultas usando una notación textual. Como parte del estándar UML estaba originalmente pensado para modelar propiedades que no podían ser expresadas de forma sencilla mediante notación gráfica, como por ejemplo, los invariantes de clases en un diagrama de clases UML. Cada expresión OCL está escrita en el contexto de un modelo (llamado *modelo contextual*) y es evaluado en un modelo de objetos (también llamado *instancia* o *escenario*) del modelo contextual. Esta evaluación devuelve un valor pero en ningún caso altera el modelo de objetos dado, debido a que la evaluación en OCL se realiza sin modificar el estado.

OCL está fuertemente tipado. Las expresiones tienen tipo primitivo, tipo clase, tipo tupla o tipo colección. OCL proporciona operadores estándares sobre datos primitivos, tuplas y colecciones. OCL proporciona también un operador punto (.) para acceder a los valores de los atributos y extremos de asociación de los objetos pertenecientes al escenario dado.

8.1.3 SecureUML

SecureUML [11] extiende Control de Acceso Basado en Roles (RBAC) [52] con *restricciones de autorización*. Estas restricciones pueden ser usadas para especificar políticas que dependen de propiedades del estado del sistema, por ejemplo, que un usuario sólo pueda escribir un mensaje en una sala de chat donde participa. De manera más específica, SecureUML permite al modelador formalizar decisiones de control de acceso dependiendo de dos tipos de información:

1. *información estática*, referida a las asignaciones de usuarios y permisos a roles y la jerarquía de roles; e

2. *información dinámica*, referida al cumplimiento de las restricciones de autorización en el estado actual del sistema.

Por lo tanto, SecureUML soporta el modelado de *roles* y sus jerarquías, *permisos*, *acciones*, *recursos* y *restricciones de autorización*. Además, permite el modelado de asignaciones: qué permisos son asignados a un rol, qué acciones son permitidas por un permiso, qué

Tabla 8.1: SecureUML+ComponentUML: acciones y recursos.

Recurso	Acciones Atómicas	Acciones Compuestas
Entidad	create, delete	read, update, full access
Atributo	read, update	full access
Método	execute	
Extremo de asociación	read, create, delete	full access

recursos son afectados por un permiso y qué restricción de autorización debe cumplirse antes de conceder un permiso.

SecureUML es, sin embargo, un lenguaje genérico ya que deja abierta la naturaleza de los recursos protegidos, a saber, si estos recursos son datos, objetos de negocio, estados de controlador, etc. En nuestra metodología usamos una extensión de SecureUML [11] que combina SecureUML con ComponentUML. En esta extensión, que por simplicidad seguiremos llamando SecureUML, los *recursos* protegidos son las entidades, junto con sus atributos, métodos y extremos de asociación, mientras que las *acciones* son las mostradas en la Tabla 2.1.

Hay dos clases de acciones: atómicas y compuestas. Las *acciones atómicas* están pensadas para ser mapeadas directamente a operaciones existentes en la capa persistente. Las *acciones compuestas* agrupan de manera jerárquica acciones atómicas de nivel inferior. Por ejemplo, la acción compuesta full access de un atributo agrupa las acciones atómicas read y update de este atributo. Finalmente, las restricciones de autorización están especificadas usando OCL, donde el contexto de una expresión OCL es el modelo de datos subyacente. Adicionalmente, las expresiones OCL en los modelos de seguridad pueden contener las variables self, caller, value y target, las cuales son interpretadas como sigue:

- self se refiere al recurso sobre el cual la acción será ejecutada, si el permiso es concedido. Notar que el recurso de un atributo, método o extremo de asociación es la entidad a la cual pertenece.
- caller se refiere al usuario que ejecutará la acción, si el permiso es concedido.
- value se refiere al valor que será usado para modificar un atributo, si el permiso es concedido.
- target se refiere al objeto que será añadido (o eliminado) de un extremo de asociación, si el permiso es concedido.

El lector familiarizado con la presentación original de SecureUML [11] notará que hemos introducido dos nuevas variables que pueden ser usadas en las restricciones de atutorización: las variables value y target. Además, para evitar potenciales ambigüedades, hemos refinado la acción update de extremos de asociación en dos acciones separadas: la acción create y la acción delete de extremos de asociación. En el Apéndice B se encuentra la definición completa de la sintaxis de SecureUML.

SecureUML proporciona varias constructoras para expresar políticas de control de acceso complejas de manera compacta e intuitiva, por ejemplo, haciendo uso de la jerarquía de acciones y roles o declarando políticas por defecto. Sin embargo, tal y como se describe en la Sección 8.3.1, todo modelo de seguridad S puede ser unívocamente transformado en un modelo $S^\flat$ semánticamente equivalente para el que se cumple lo siguiente:

Remark 3 *Sea S un modelo de seguridad, para toda acción atómica act y todo rol r en S, existe exactamente un permiso en $S^\flat$ (posiblemente con la restricción* false*) en r para ejecutar act.*

Informalmente, el modelo $S^\flat$ hace completamente explícita la política de seguridad especificada en S. Por consiguiente, sea *Auth* la función que, para todo modelo de seguridad S, rol r y acción *act*, devuelve la restricción de autorización asociada al único permiso que está definido en $S^\flat$, del rol r, para ejecutar *act*. Usaremos esta función *Auth* para definir la transformación de modelos que, en nuestra metodología, traslada la política de seguridad desde el modelo de seguridad al modelo de GUI.

8.2 Modelos de GUI

Los modelos de GUI proporcionan una vista de un sistema orientada a la interfaz humana. Junto con los modelos de datos, constituyen modelos de aplicaciones independientes de plataforma omitiendo los aspectos relacionados con la seguridad.

De manera informal, una GUI está compuesta de widgets: elementos visuales que muestran información y ejecutan acciones. En esta Sección presentamos un componente clave de nuestra metodología: un innovador lenguaje para modelar GUIs de aplicaciones de manejo de datos llamado GUI Modeling Language (GUIML, de sus siglas en inglés). En el Apéndice C se encuentra la definición completa de la sintaxis de GUIML. Es importante, sin embargo, entender que GUIML es un lenguaje para modelar no sólo la *estructura* de una GUI, es decir, los elementos (widgets) que la componen, sino también el *comportamiento* de la GUI, es decir, cómo los elementos reaccionarán (acciones) en respuesta a la interacción de los usuarios con ellos (eventos). De hecho, la característica fundamental de GUIML es el lenguaje que proporciona para modelar el comportamiento de la GUI, el cual usa OCL para especificar las condiciones y los argumentos de las diferentes acciones. Esta característica permite a los modelos de seguridad y de GUI "hablar" el mismo lenguaje (a saber, OCL en el contexto del modelo de datos común subyacente). Esto nos permite definir de manera rigurosa la función de transformación que traslada la política de seguridad al nivel de GUI.

A continuación describimos los principales elementos de GUIML, a saber, *widgets* (con sus *variables* asociadas), *eventos* y *acciones*.

Widgets

Un modelo de GUI consta de widgets de diferentes tipos: ventanas (páginas, cuando se refiere a aplicaciones web), desplegables (listas seleccionables), tablas, campos de fecha, cambos booleanos (cajas seleccionables), botones, campos de texto y etiquetas. Los widgets pueden ser mostrados en *contenedores*, los cuales también son widgets. Los widgets distintos a las ventanas deben estar contenidos en otro widget. Sólo las ventanas, los desplegables y las tablas pueden contener otros widgets. Los widgets pueden contener variables (las cuales almacenan valores para su posterior uso) y disparar eventos (los cuales ejecutan acciones).

Variables

Cada declaración de un widget puede contener declaraciones de variables, listando las variables que el widget posee.

Además de las variables explícitamente declaradas, hay variables que existen por defecto que todo widget de un tipo dado posee. Estas variables son declaradas de manera

implítica en toda declaración de widget y sus valores almacenados son manejados de maneras especiales. La descripción de cada una de esta variables, se encuentra en la página web de ActionGUI [1].

Eventos

Cada declaración de widget puede contener declaraciones de eventos. Los eventos son disparados cuando el usuario interactúa con los widgets de la GUI, ejecutando acciones sobre datos o sobre otros widgets.

Las acciones ejecutadas cuando un evento se dispara son especificadas mediante el uso de *sentencias*. Una sentencia puede ser una acción, una sentencia condicional, una iteración o una secuencia de sentencias. En GUIML, las condiciones en las sentencias condicionales y en las iteraciones son especificadas mediante el uso de expresiones OCL, cuyo contexto es el modelo de datos subyacente. De manera adicional, estas expresiones pueden referirse a variables de widgets. Cada secuencia de sentencias asociada a un evento es ejecutada como una simple *transacción*: o todas sus sentencias son ejecutadas con éxito en el orden dado o ninguna de ellas es ejecutada.

Acciones

Toda declaración de evento contiene una secuencia de sentencias que especifica las acciones ejecutadas cuando el evento es disparado. Estas acciones pueden ser ejecutadas sobre objetos que pertenecen a la capa persistente o sobre objetos que pertenecen a la capa de presentación. En el primer caso se llaman *acciones sobre datos* y en el segundo caso se llaman *acciones de GUI*. Mencionar que algunas acciones pueden tomar argumentos cuyos valores son conocidos únicamente en tiempo de ejecución, por ejemplo, una acción de borrar (delete) cuyo argumento es el elemento seleccionado por el usuario en un desplegable o una acción de actualización (update) cuyo argumento es el número introducido por el usuario en una caja de texto. En GUIML, estos valores se especifican mediante el uso de expresiones OCL. De nuevo, el contexto de estas expresiones OCL es el modelo de datos subyacente y también pueden referirse a variables de widgets.

A continuación, describimos brevemente de manera informal la semántica de algunas de las acciones de datos de GUIML.

- **Entity create:** Crea una instancia de una entidad en la capa persistente.
- **Entity delete:** Borra una instancia de una entidad de la capa persistente.
- **Attribute read:** Lee el valor de un atributo de una instancia de la capa persistente.
- **Attribute update:** Modifica el valor de un atributo de una instancia de la capa persistente.
- **Association-end read:** Lee la colección de instancias asociadas al extremo de asociación de una instancia de la capa persistente.
- **Association-end create:** Crea un enlace entre dos instancias de la capa persistente a través de un extremo de asociación.
- **Association-end delete:** Borra un enlace entre dos instancias de la capa persistente a través de un extremo de asociación.

Finalmente, describimos brevemente de manera informal la semántica de algunas de las acciones de GUI definidas en GUIML.

- **Set:** Modifica el valor de una variable de widget.
- **Open:** Abre una ventana. Adicionalmente, puede tomar como argumentos parejas ($variable_i$, $valor_i$) donde $variable_i$ es el nombre de una variable que pertenece a la ventana destino y $valor_i$ es el valor que será asignado a $variable_i$ cuando la ventana destino sea abierta.
- **Back:** Vuelve a la ventana anterior en el historial de navegación.
- **Fail:** Deshace la transacción actual, por lo que la sentencia actual no se ejecuta de manera satisfactoria.
- **Skip:** No hace nada.

8.3 Modelos de GUI con Seguridad

8.3.1 Política de seguridad explícita

En esta Sección definimos una transformación que, para cada modelo de seguridad S, produce el modelo de seguridad $S^\flat$ el cual hace explícita la política de seguridad declarada en S. Definimos esta transformación en cuatro pasos. Tal y como se declara en la Definición 3, al final de nuestra transformación se cumple que: para cada acción atómica *act* y cada rol r en S, existe exactamente un permiso en $S^\flat$ (posiblemente restringido por false) en r para ejecutar *act*.
Paso 1: Copiar los permisos explícitos

- *Acciones atómicas.* Sea *act* una acción atómica. Supongamos que existe un permiso en S de un rol r para ejecutar *act* bajo una restricción *auth*. Entonces también existe un permiso en $S^\flat$ del rol r para ejecutar *act* bajo la misma restricción *auth*.

Paso 2: Desplegar el modelo de seguridad

- *Jerarquía de acciones.* Sea *CA* una acción compuesta. Supongamos que existe un permiso en S del rol r para ejecutar *CA* bajo una restricción *auth*. Entonces para cada acción atómica *act* contenida en *CA* existe un nuevo permiso en $S^\flat$ del rol r para ejecutar *act* bajo la misma restricción de autorización *auth*.
- *Jerarquía de roles.* Sea *act* una acción atómica, y sean r y r' dos roles. Supongamos que r es un sub-rol de r' en S y que también existe un permiso en S del rol r' para ejecutar *act* bajo la restricción *auth*. Entonces existe un nuevo permiso en $S^\flat$ del rol r para ejecutar *act* bajo la misma restricción de seguridad *auth*.
- *Acciones de borrar (delete).* Sea *entity* una entidad. Supongamos que existe un permiso en S del rol r para borrar una instancia de la entidad *entity* bajo una restricción *auth*. Entonces para cada extremo de asociación *assoc* perteneciente a *entity*, existe un nuevo permiso en $S^\flat$ del rol r para ejecutar la acción **Delete::***assoc* bajo la misma restricción *auth*.
- *Extremos de asociación opuestos.* Sean *assoc* y *assoc*$'$ dos extremos de asociación opuestos. Sea *act* una acción **Create::***assoc*. Supongamos que existe un permiso en S

del rol r para ejecutar act bajo la restricción $auth$. Entonces existe un nuevo permiso en $S^\flat$ del rol r para ejecutar la acción **Create::**$assoc'$ bajo la restricción que resulta de reemplazar simultáneamente en $auth$ la variable self por target y la variable target por self. El despliegue es similar cuando act es la acción **Delete::**$assoc$.

Paso 3. Añadir permisos por defecto al modelo de seguridad

- *Denegación por defecto.* Sea r un rol y sea act una acción atómica. Supongamos que no existe un permiso en $S^\flat$ del rol r para ejecutar act. Entonces existe un nuevo permiso en $S^\flat$ del rol r para ejecutar act bajo la restricción false. Esto significa que el rol r tendrá denegado el acceso para ejecutar act en todas las circunstancias.

Paso 4. Simplificar el modelo de seguridad resultante

- *Disyunción de restricciones.* Sea r un rol y sea act una acción atómica. Supongamos que existen n permisos en $S^\flat$ del rol r para ejecutar act. Entonces estos n permisos son simplificados en un único permiso, cuya restricción de autorización resulta de la disyunción de todas las restricciones de autorización de cada uno de los n permisos individuales.

8.3.2 Modelos de GUI seguros

En esta Sección describimos el corazón de nuestra metodología: una función de transformación de modelos *Sec* que, dado un modelo de GUI G y un modelo de seguridad S, genera automáticamente un nuevo modelo GUI $\mathrm{Sec}(G, S)$. El modelo generado es idéntico a G excepto que es *seguro* con respecto a S. La función de transformación *Sec* funciona envolviendo alrededor de cada acción de datos act en G una sentencia "if-then-else" con los siguientes argumentos:

- una condición que refleja las restricciones asociadas a los permisos especificados en S, para cada uno de los diferentes roles, para ejecutar la acción act;
- una rama "then" que contiene la acción act y
- una rama "else" que contiene la acción **fail**.

Por lo tanto, la semántica de la sentencia "if-then-else" asegura que act sólo será ejecutada,si se cumplen las restricciones asociadas a los permisos correspondientes. Además, esta semántica garantiza que si estas restricciones no se cumplen, entonces la acción **fail** será ejecutada deshaciendo la transacción actual.

De manera más específica, para generar la sentencia "if-then-else" anteriormente mencionada, la función *Sec* hace uso de la Definición 3. En particular, para cada rol r en S, *Sec* llama a la función $Auth(S, r, act)$ para obtener la expresión que finalmente (es decir, cuando la política de seguridad se ha hecho completamente explícita) restringe el permiso dado por r para ejecutar act. Sin embargo, ya que esta expresión puede contener las variables self, value, target y caller, la función *Sec* debe también reemplazar estas variables por los actuales argumentos de la acción act (incluyendo su usuario actual). Denotamos la expresión OCL resultante por $\mathrm{Auth}(S, r, act)[args]$ donde $args$ son los argumentos especificados en el modelo GUI por la acción act. Finalmente, ya que diferentes roles pueden estar restringidos por diferentes expresiones, la condición generada por *Sec* tendrá la forma:

((
r_1 = [*Window*.role] and Auth(S, r_1, *act*)[*args*])
or ... or
(r_n = [*Window*.role] and Auth(S, r_n, *act*)[*args*])),

donde $r_1, \ldots, r_n$ son todos los roles declarados en S. El actual usuario de la aplicación y su rol están almacenados en las variables de widget caller y role, las cuales pertenecen a cada una de las ventanas en el modelo de GUI.

8.3.3 Corrección de la Transformación de Modelos

Resumimos aquí la corrección de la transformación de modelos *Sec*, la cual se define de manera relativa a la semántica de los modelos de GUI. Todos los detalles se encuentran en [6]. Definimos la semántica de los modelos de GUI dando primero un conjunto de reglas de inferencia que define una relación de transición $\longrightarrow$ entre triplas de la forma $\langle stm, I, \theta \rangle$, donde *stm* es una sentencia, I es un escenario (es decir, una instancia del modelo de datos subyacente) y θ representa un estado de las variables de widget. Proporcionamos reglas de inferencia para cada posible sentencia: para cada tipo de acción de datos y de acción de GUI (casos base) y para secuencias arbitrarias de sentencias, sentencias condicionales y sentencias de iteración (casos inductivos). En particular, para acciones de datos la regla de inferencia tiene la forma

$$\frac{}{\langle act(args), I, \theta \rangle \longrightarrow \langle \textbf{skip}, \mathrm{res}(I), \mathrm{res}(\theta) \rangle} \,,$$

donde:

- *args* son los argumentos de la acción de datos *act*;
- res(I) especifica el escenario que resulta de ejecutar $act(args)$ en el escenario I con el estado de variables de widget θ y
- res(θ) especifica el nuevo estado de las variables de widget tras la ejecución de $act(args)$ en el escenario I y estado de las variables de widget θ.

De manera crucial, no se define ninguna regla de inferencia que lleve a **skip** para la acción de GUI **fail**.

Luego definimos la *semántica operacional* de un evento *ev* que ejecuta las acciones especificadas por la sentencia *stm* como el conjunto de todas las transiciones

$$\langle stm, I, \theta \rangle \longrightarrow^* \langle \textbf{skip}, I', \theta' \rangle \,,$$

donde $\longrightarrow^*$ es la clausura reflexiva-transitiva de $\longrightarrow$.

Por definición, esta semántica de operaciones para eventos es *sin seguridad*: no respeta las restricciones de seguridad que, de acuerdo con el modelo de seguridad dado, deben restringir la ejecución de las acciones de datos. Para proporcionar una semántica operacional para eventos *con seguridad* definimos la *versión segura* de las reglas de inferencia. En particular, dado un modelo de seguridad S, para cada rol r y para cada tipo de acción de datos *act*, la versión segura de la regla de inferencia para *act* y r tiene la forma

$$\frac{[\![(\mathrm{Auth}(S, r, act)[args])\theta]\!]^I = \mathsf{true}}{\langle act(args), I, \theta \rangle \longrightarrow \langle \textbf{skip}, \mathrm{res}(I), \mathrm{res}(\theta) \rangle} \,,$$

donde:

- $[\![expr]\!]^I$ denota el valor de la expresión *expr* en el escenario I y por lo tanto;
- $[\![(\text{Auth}(S, r, act)[args])\theta]\!]^I$ denota la evaluación en el escenario I de la autorización $\text{Auth}(S, r, act)$ que restringe el único permiso que, de acuerdo con la Definición 3, finalmente permite a los usuarios con rol r ejecutar la acción *act*, siendo *arg* los argumentos de *act* y θ el estado de las variables de widget.

Estas reglas de inferencia con seguridad definen la relación de transición $\longrightarrow_S$. Finalmente, dado un modelo de seguridad S definimos la *semántica operacional con seguridad* de un evento *ev* que ejecuta las acciones especificadas por una sentencia *stm* como el conjunto de todas las transiciones

$$\langle stm, I, \theta\rangle \longrightarrow_S^* \langle \textsf{skip}, I', \theta'\rangle\,.$$

El siguiente teorema formaliza la *corrección* de nuestra función de transformación de modelos *Sec*. Declara que la evaluación de una sentencia transformada por *Sec*, siguiendo la semántica operacional sin seguridad, devuelve el mismo resultado que la evaluación de la sentencia original usando la semántica operacional con seguridad. Por lo tanto, la sentencia transformada respeta la restricción de autorización formalizada en el modelo de seguridad subyacente.

Teorema Sea S un modelo de seguridad y sea *stm* una sentencia. Entonces para cada escenario I y cada estado de variables de widget θ.

$$\begin{aligned}\langle \text{Sec}(stm, S), I, \theta\rangle \longrightarrow^* \langle \textsf{skip}, I', \theta'\rangle \Longleftrightarrow \\ \langle stm, I, \theta\rangle \longrightarrow_S^* \langle \textsf{skip}, I', \theta'\rangle.\end{aligned}$$

Capítulo 9

Soporte para el desarrollo dirigido por modelos

Los modelos de GUI con seguridad son independientes de plataforma y pueden ser mapeados a implementaciones que emplean diferentes tecnologías. Esto incluye aplicaciones de escritorio, aplicaciones web y aplicaciones para móviles. Como parte de nuestro trabajo construimos el conjunto de herramientas ActionGUI [1], el cual genera de manera automática aplicaciones web de gestión de datos a partir de modelos de GUI con seguridad.

Tras una breve descripción del estado actual del conjunto de herramientas ActionGUI, en este Capítulo describimos nuestra experiencia en el desarrollo del evaluador "Eye OCL Software" (EOS, de sus siglas en inglés). EOS es un componente Java para la evaluación eficiente de OCL en escenarios mediano-grandes. Comparamos EOS con otros evaluadores de OCL en escenarios mediano-grandes y exploramos varias estrategias de evaluación de expresiones OCL en escenarios realmente grandes. Luego proponemos un mapeo de un subconjunto de OCL a lógica de primer orden (FOL) y usamos dicho mapeo para comprobar la insatisfacibilidad de conjuntos de restricciones OCL. Argumentamos que nuestro mapeo es simple, ya que las sentencias FOL resultantes se asemejan mucho a las restricciones OCL originales, y práctico, ya que podemos usar herramientas de razonamiento automático, tales como demostradores automáticos de teoremas y resolutores SMT, para comprobar de forma automática la insatisfacibilidad de conjuntos no triviales de restricciones OCL. Finalmente, proponemos una metodología para el razonamiento sobre políticas de control de acceso de grano fino, cuyas restricciones de autorización son especificadas en OCL, mediante el uso del mapeo de OCL a FOL anteriormente mencionado.

9.1 El conjunto de herramientas ActionGUI

El conjunto de herramientas ActionGUI dispone de editores de modelos para construir y manipular modelos de datos, de seguridad y de GUI. Estos editores comparten nuestro propio analizador sintáctico de OCL, el cual toma como entrada adicional las variables introducidas por los distintos modelos junto con sus respectivos tipos: en el caso de los modelos de seguridad las variables self, caller, target y value, y en el caso de los modelos de GUI todas las variables de widget dadas. De forma crucial, el conjunto de herramientas ActionGUI implementa nuestra transformación de modelos para generar modelos de GUI con seguridad. Finalmente, incluye un generador de código que, dado un modelo de GUI con seguridad, produce una aplicación web basada en la siguiente arquitectura estándar de tres capas.

1. Capa de presentación (también conocida como front-end): Los usuarios acceden a las aplicaciones web a través de navegadores estándares, los cuales renderizan el contenido (HTML y JavaScript) proporcionado por el servidor web de forma dinámica.

2. Capa de aplicación: El conjunto de herramientas genera Aplicaciones Web Java implementadas mediante el uso de la librería Vaadin [82]. Las aplicaciones corren en un servidor web (como Tomcat o GlassFish), procesan las solicitudes de los clientes y generan contenido el cual es enviado de vuelta al cliente para ser renderizado. También pueden manipular datos almacenados en la capa persistente. Cuando las solicitudes de los clientes son procesadas la aplicación generada *interpreta* su modelo de GUI con seguridad subyacente. En particular, lleva a cabo las comprobaciones de seguridad requeridas antes de modificar algún dato almacenado en la capa persistente o antes de enviar algún dato a la capa persistente.

3. Capa persistente (también conocida como capa de datos o back-end): La aplicación generada gestiona la información almacenada en una base de datos. Para cada aplicación, el conjunto de herramientas genera el correspondiente esquema de la base de datos a partir del modelo de datos de la aplicación. Dicho esquema contiene los comandos que crean la base de datos.

El conjunto de herramientas ActionGUI produce sus mejores resultados cuando la funcionalidad de la aplicación de gestión de datos pretendida se puede reducir a acciones de tipo CRUD y su dinámica consiste en navegar entre ventanas e intercambiar información con la base de datos subyacente. Para aplicaciones en esta categoría, ActionGUI genera de forma automática la implementación completa a partir de los modelos de datos, de seguridad y de GUI. Las llamadas a acciones de tipo CRUD son modeladas en GUIML mediante el uso de acciones sobre datos y la navegación y el paso de información entre ventanas es modelado mediante el uso de acciones de GUI: **open**, **back** y **set**. Por supuesto, algunas aplicaciones de gestión de datos pueden requerir funcionalidad adicional, como por ejemplo, la posibilidad de enviar correos electrónicos, imprimir tablas o exportar datos en formatos específicos. Como es de esperar, el conjunto de herramientas ActionGUI no genera código para estos métodos no-CRUD. En su lugar, incluye su implementación — la cual debe ser proporcionada por el desarrollador de la aplicación — en la aplicación generada de manera que cuando la aplicación necesita interpretar alguno de estos métodos simplemente llama al método provisto.

9.2 Evaluación de expresiones OCL

En [33] presentamos nuestra experiencia en el desarrollo del evaluador Eye OCL Software (EOS), un componente Java para la evaluación eficiente de OCL. Primero motivamos la necesidad de una implementación eficiente de OCL para hacer frente a los novedosos usos del lenguaje. Luego analizamos algunos aspectos que, basados en nuestra experiencia, deberían tenerse en cuenta a la hora de construir un evaluador OCL para escenarios de tamaño mediano-grande. Finalmente, exploramos algunas estrategias de evaluación de expresiones OCL en escenarios realmente grandes.

9.2.1 Motivación

Como parte de nuestra investigación, hemos buscado distintos usos de OCL más allá de su "requisito inicial de lenguaje de modelado preciso como complemento de las especificaciones UML". Dos tipos de usos de OCL relacionados entre sí han atraído nuestra

atención [16, 3, 4, 35, 8], ambos tienen que ver con el uso de OCL para analizar modelos definidos por el usuario mediante la evaluación de consultas sobre las correspondientes instancias de nuestros metamodelos. Debido a que estas instancias suelen contener un gran número de elementos, la evaluación de expresiones sobre ellos conlleva un alto coste computacional.

Consideremos, por ejemplo, el uso de OCL para expresar métricas de programas Java (miembros del grupo Triskell en IRISA, Francia, nos sugirieron este uso de OCL). Los escenarios en los cuales las métricas del programa serán evaluadas son las instancias del metamodelo de Java correspondiente a los programas: por lo tanto, cuanto más largos son los programas más largos son los escenarios[1] y consecuentemente mayor es el coste computacional de evaluar las métricas del programa.

En [33] describimos nuestra experiencia en el desarrollo del componente Eye OCL Software [41] (EOS, de sus siglas en inglés), un evaluador de OCL diseñado con el objetivo de realizar una evaluación eficiente de expresiones OCL sobre escenarios de tamaño mediano-grande. En particular, analizamos i) la necesidad de una implementación eficiente de OCL para cubrir los distintos usos del lenguaje; ii) los aspectos que hemos tenido que tener en cuenta para mejorar la eficiencia del evaluador EOS en escenarios de tamaño mediano-grande y iii) los límites de las actuales implementaciones de OCL sobre escenarios de tamaño realmente grande. Para ello, en [33] analizamos los resultados de aplicar un banco de pruebas sobre varios evaluadores de OCL (incluyendo el evaluador EOS) con los objetivos de: i) mostrar el rendimiento de estos evaluadores sobre escenarios de tamaño mediano-grande y ii) de ilustrar los aspectos que consideramos que deben tenerse en cuenta a la hora de implementar un evaluador OCL eficiente sobre escenarios de tamaño mediano-grande. De forma interesante, esta calidad — la eficiencia del motor OCL sobre escenarios de tamaño mediano-grande — no es cubierta en el banco de pruebas propuesto en [54]: de hecho, el escenario más grande para la comprobación de la eficiencia del motor OCL propuesto en [55],contiene sólo 42 objetos y 42 enlaces entre ellos. A pesar de los resultados de nuestro banco de pruebas esta Sección no es una propaganda del evaluador EOS: como "producto", el evaluador está aun en su infancia.

9.2.2 Medidas de coste de evaluación de expresiones

Aunque el coste computacional de evaluar una expresión OCL particular en un escenario dado obviamente depende de los algoritmos y las estructuras de datos usadas en cada herramienta, basándonos en nuestra experiencia existen dos tipos de medidas que merecen la pena ser consideradas antes de ejecutar la evaluación de una expresión: primero, el máximo número de veces que se accede a las propiedades de un objeto y segundo, el tamaño máximo de las colecciones que serán construidas. En el caso de escenarios de tamaño mediano-grande el reto para los motores de OCL es que los valores de estas medidas son típicamente grandes. El uso de los tipos de medidas anteriormente mencionadas nos permite comprobar el rendimiento del evaluador EOS, así como buscar posibles optimizaciones.

9.2.3 La implementación del evaluador EOS

ITP/OCL [31] es un evaluador de OCL basado en reescritura, implementado a partir de una semántica ecuacional ejecutable para OCL [49]. Aunque esta herramienta rinde ra-

[1] Como ejemplo, la aplicación SpoonEMF desarrollada por el grupo Triskell genera, para un programa estándar de Java de 10 líneas de código, un escenario con 113 objetos; para un programa de 100 líneas, uno con 1180 objetos y para un programa de 500 líneas, uno con 3470 objetos.

zonablemente bien sobre escenarios de tamaño pequeño-mediano, su rendimiento no escala en el caso de escenarios de tamaño mediano-grande. Provocado por nuestro interés sobre aplicaciones que requieren una evaluación eficiente de OCL sobre escenarios de tamaño mediano-grande, decidimos implementar el evaluador Eye OCL Software (EOS). EOS es un componente Java cuyo diseño sigue las ideas clave que hay detrás de la herramienta ITP/OCL. Su implementación incluye un analizador sintáctico (parser) de OCL (el cual hace uso de la librería SableCC [53]) y un evaluador OCL (implementado en unas 7K líneas de código Java). EOS soporta la Librería Estándar de OCL 2.0 [72], casi en su totalidad, con la excepción del tipo OclMessage y sus operaciones. Con la idea de hacerlo lo más general posible, el componente EOS no está basado en un entorno de desarrollo sobre (meta)modelos en particular: su interfaz pública proporciona métodos para insertar elementos, uno a uno, en los modelos y escenarios del usuario y para insertar expresiones al evaluador como cadenas de caracteres ASCII. Esta decisión también nos permitió diseñar la estructura de datos de EOS para almacenar internamente los modelos y escenarios del usuario de tal manera que el acceso a las propiedades de los objetos fuera eficiente. La otra posible novedad en su implementación es que, antes de evaluar una expresión de tipo collect, intentamos (sobre)estimar el tamaño de la colección resultante para asignar la memoria por adelantado. Es resto de la implementación es bastante directa: las expresiones iteradoras de OCL son implementadas por bucles while/for en Java y las operaciones estándar de OCL, cuando es posible, se implementan usando los operadores Java correspondientes. Como es de esperar, EOS evalúa las expresiones siguiendo una estrategia impaciente: en particular, las expresiones de tipo colección son evaluadas en su totalidad asignando en memoria todos los elementos resultantes.

9.2.4 Limitaciones

Para evaluar expresiones en escenarios realmente grandes necesitamos primero resolver el problema de cargar los escenarios en los evaluadores OCL. Para ello, hemos explorado dos estrategias diferentes ambas basadas en la representación de los modelos y escenarios del usuario como bases de datos relacionales. La primera estrategia consiste en modificar el evaluador OCL de forma que busque la información contenida en los escenarios directamente en su representación en la base de datos. La ventaja de esta estrategia es que sólo requiere modificar la evaluación de las expresiones de punto (expresiones de acceso a propiedades de objetos) de la forma esperada: el acceso al valor de un atributo o un extremo de asociación de un objeto será ahora implementado como una consulta `select` básica de SQL. La forma concreta de estas consultas depende, por supuesto, del mapeo usado para representar modelos como bases de datos relacionales (ver, por ejemplo, [75, 80] y [56]). Para comprobar la viabilidad de esta estrategia realizamos las modificamos correspondientes en el evaluador EOS: desafortunadamente, el coste de evaluar expresiones de punto a través del uso del controlador JDBC era tan alto que hacía impracticable el uso del evaluador EOS modificado, a la hora evaluar expresiones en escenarios realmente grandes.[2]

La segunda estrategia consiste en traducir las expresiones OCL a expresiones en un lenguaje de consultas ya disponible para las bases de datos relacionales. Según nuestros conocimientos, los resultados más interesantes en esta línea son analizados en [47, 58] y proporcionan los fundamentos de la herramienta OCL2SQL [57, 81]. Sin embargo, la solución propuesta en [47, 58] no es del todo satisfactoria. Primero, sólo considera un subconjunto restringido de OCL: en particular, no maneja iteradores ni tuplas ni colecciones.

[2]Para este experimento, mapeamos cada clase del modelo a una tabla cuyas columnas corresponden a sus atributos y mapeamos cada asociación a una tabla cuyas columnas corresponden a sus extremos de asociación.

Segundo, sólo se aplica sobre expresiones de tipo Boolean y no sobre consultas arbitrarias. Finalmente, la complejidad de las consultas SQL obtenidas de esta traducción es tan alta que hace impracticable el uso de la evaluación OCL sobre escenarios de tamaño realmente grande. [3]

9.3 Comprobación de insatisfacibilidad de expresiones OCL

En [34] proponemos un mapeo de un subconjunto de OCL a lógica de primer orden (FOL) y usamos este mapeo para comprobar la insatisfacibilidad de conjuntos de restricciones OCL. En [34] argumentamos que nuestro mapeo es simple, ya que las sentencias FOL resultantes son muy similares a las restricciones OCL originales, y práctico, ya que podemos usar herramientas de razonamiento automático, tales como demostradores automáticos de teoremas y resolutores de Satisfacibilidad Módulo Teorías (SMT, de sus siglas en inglés), para comprobar de manera automática la insatisfacibilidad de conjuntos no triviales de restricciones OCL. SMT generaliza la satisfacibilidad booleana (SAT) incorporando razonamiento de igualdad, aritmética, vectores de bits de tamaño fijo, arrays, cuantificadores y otras útiles teorías de primer orden [5].

9.3.1 Motivación

La falta de soporte de herramientas para OCL fue señalado en [29] como la principal causa de la limitada adopción del lenguaje en la industria. Desde entonces, varias iniciativas han tenido éxito en este sentido y sus resultados están disponibles para los modeladores (ver [46]).

El trabajo presentado en [34] propone un mapeo de OCL a lógica de primer orden, el cual es definido con el propósito de soportar comprobaciones de insatisfacibilidad *sin límites*, de expresiones OCL mediante el uso de herramientas de razonamiento *automático*. Bajo nuestro punto de vista, poder comprobar la *insatisfacibilidad* de (conjuntos de) expresiones OCL es una herramienta potente ya que permite a los modeladores (entre otras tareas):

- verificar invariantes de clases, comprobando que lógicamente implican las propiedades/restricciones esperadas;
- verificar pre-condiciones de métodos, comprobando que las invariantes de clases no implican lógicamente sus negaciones y
- verificar post-condiciones de métodos, comprobando que no implican lógicamente la negación de (alguna de) las invariantes de clases.

Sin embargo, bajo nuestro punto de vista, lo que hace a un resolutores de insatisfacibilidad no sólo potente, sino también práctico, es que sea *automático*. Dada la naturaleza indecidible del lenguaje OCL completo, se espera tener un comprobador de insatisfacibilidad automático para un gran tipo de expresiones OCL. En [34] no tratamos de definir cómo de grande y/o interesante es la clase de expresiones OCL insatisfacibles que podemos comprobar automáticamente. Sin embargo, argumentamos que nuestro mapeo es simple, ya que las sentencias FOL resultantes son muy similares a las restricciones OCL originales, y práctico, ya que podemos usar demostradores de teoremas automáticos (como por

[3] El principal desarrollador de la herramienta OCL2SQL nos confirmó que la eficiencia de la herramienta OCL2SQL no había sido probada en escenarios de tamaño realmente grande (comunicación por correo electrónico, mayo del 2008).

ejemplo Prover9 [67]) y resolutores SMT (como por ejemplo Yices [48]), para comprobar automáticamente la insatisfacibilidad de conjuntos de restricciones de OCL no triviales.

9.3.2 Insatisfacibilidad de restricciones OCL

La noción de *insatisfacibilidad* que proponemos enfatiza el significado lógico de las restricciones OCL, de hecho, básicamente traduce para OCL la noción estándar de insatisfacibilidad para fórmulas de primer orden.

En lo que sigue, denotamos *restricción* OCL a cualquier expresión OCL de tipo Boolean. Asumimos que las instancias de los modelos no siempre tienen un número finito de elementos.

Definition 2 *Dado un modelo (diagrama de clases) $\mathcal{M}$ y un conjunto de restricciones OCL Φ, decimos que Φ es* $\mathcal{M}$-insatisfacible *si y sólo si no existe una $\mathcal{M}$-instancia (diagrama de objetos) $\mathcal{O}$ sobre el cual toda restricción en Φ evalúa a* true.

Otras nociones de satisfacibilidad/insatisfacibilidad de modelos UML con restricciones OCL pueden encontrarse en la literatura. En particular, las nociones usadas en [23, 24] son las nociones de *satisfacibilidad débil* y *satisfacibilidad fuerte* (y otras nociones relacionadas son también introducidas en [18]). Satisfacibilidad débil significa que existe un número finito de instancias del modelo en las cuales al menos una clase contiene al menos un elemento. Satisfacibilidad fuerte significa que existe un número finito de instancias del modelo en las cuales todas sus clases tienen al menos un elemento. Si un conjunto de restricciones es insatisfacible (en nuestro sentido) entonces no puede ser ni débilmente satisfacible ni fuertemente satisfacible. Por otro lado, si un conjunto de restricciones no es ni débilmente satisfacible ni fuertemente satisfacible, no implica que sea instatisfacible (en nuestro sentido). Esto se debe a que la instancia vacía del modelo, en la que ninguna clase contiene algún elemento, podría ser satisfacible (en nuestro sentido) siempre y cuando se cumplan todas las restricciones.

9.3.3 Un mapeo de OCL a FOL

En esta Sección proporcionamos una descripción de alto nivel de un mapeo de un subconjunto de OCL a lógica de primer orden (FOL). El lector interesado puede encontrar en [34] todos los detalles de esta definición. Dado un conjunto de restricciones OCL, nuestro mapeo genera un conjunto de fórmulas FOL tales que si el conjunto resultante es insatisfacible, entonces el conjunto original también es insatisfacible.

Las restricciones OCL especifican propiedades que un modelo debe satisfacer. Para poder hacer esto de manera concisa, OCL proporciona diferentes constructoras para referirse a colecciones de elementos específicos. De forma breve, nuestro mapeo está definido de manera recursiva sobre la estructura de las expresiones OCL:

- Las expresiones de tipo Boolean se traducen a fórmulas (las cuales esencialmente reflejan su estructura lógica), las expresiones de tipo Integer básicamente son copiadas y, en este punto, no mapeamos expresiones de tipo String.
- Las expresiones de tipo Collection se traducen a predicados cuyos significados se definen mediante fórmulas adicionales generadas por el mapeo y, en este punto, sólo mapeamos expresiones de colección de tipo Set.
- Los extremos de asociación se traducen a predicados los cuales son también definidos mediante fórmulas adicionales generadas por el mapeo y, en este punto, no mapeamos asociaciones cualificadas.

- Los atributos se traducen a funciones, las cuales no son interpretadas por el mapeo.

Una ventaja crucial de nuestro mapeo es que la insatisfacibilidad de las fórmulas resultantes puede ser comprobada mediante el uso de herramientas de razonamiento *automático*, tales como demostradores automáticos de teoremas y resolutores SMT. Para ilustrar esta idea, en [34] presentamos una serie de resultados obtenidos a partir de probar (de manera automática) la insatisfacibilidad de diferentes subconjuntos de restricciones OCL. Para ello, usamos dos herramientas: Prover9 [67] (un demostrador automático de teoremas) y Yices [48] (un resolutor SMT). Ambas herramientas terminaron cada una de las pruebas en menos de un segundo usando un ordenador portátil estándar.

9.3.4 Limitaciones

El mapeo de OCL a lógica de primer orden que aquí se presenta soporta el uso directo de resolutores SMT para comprobar la satisfacibilidad de expresiones OCL, pero bajo restricciones específicas sobre las expresiones que son permitidas y sobre las instancias de los modelos de datos (escenarios) que son comprobados. No sorprende que algunas de estas restricciones sean impuestas para evitar el problema de tratar la indefinibilidad en OCL. En particular, nuestro mapeo i) sólo considera instancias (objetos) cuyos atributos están definidos y ii) no permite expresiones que contengan operadores que puedan generar valores indefinidos. En [39] se proponen modificaciones al mapeo de OCL a lógica de primer orden que, a pesar de estar basado en los mismos principios que nuestro mapeo subyacente, está diseñado para tratar las limitaciones del anterior mapeo que tienen que ver con la indefinibilidad en OCL. A continuación explicamos de forma breve la diferencia clave entre ambos mapeos. Bajo las restricciones i) y ii) descritas arriba, una expresión OCL de tipo Boolean sólo puede evaluarse a true o false. Por lo tanto, para razonar en este contexto mediante el uso de expresiones de tipo Boolean basta con definir un mapeo que formalice cuándo una expresión OCL de tipo Boolean se evalúa a true. Esto es precisamente lo que hacemos en nuestro mapeo aquí presentado. Sin embargo, en presencia de la indefinibilidad las expresiones OCL de tipo Boolean también pueden ser evaluadas a null o invalid. Para tratar este hecho, [39] define cuatro mapeos diferentes los cuales formalizan, respectivamente, cuándo una expresión OCL de tipo Boolean se evalúa a true, cuándo a false, cuándo a null, y cuándo a invalid.

9.4 Razonamiento sobre políticas de control de acceso

En esta Sección presentamos una metodología novedosa y con soporte de herramienta para el razonamiento sobre políticas de control de acceso de grano fino (FGAC). Además, describimos nuestra experiencia con respecto al uso del resolutor SMT Z3 [45] para probar de forma automática propiedades no triviales sobre políticas FGAC.

El componente clave de nuestra metodología es el mapeo [34, 39] de OCL a lógica de primer orden presentado en la Sección 9.3.3, el cual permite transformar cuestiones sobre políticas FGAC a problemas de satisfacibilidad en lógica de primer orden. A pesar de que este mapeo no cubre el lenguaje OCL al completo, nuestra experiencia muestra que el tipo de expresiones OCL que típicamente se usa para especificar invariantes y restricciones de autorización son cubiertas por el mapeo. Más interesante es, sin embargo, la efectividad de los resolutores SMT para razonar automáticamente sobre políticas FGAC. A pesar de que nuestra experiencia es extremadamente alentadora (todos los problemas son resueltos en menos de un segundo), no debemos olvidar que nuestros resultados dependen completamente de la interacción entre i) la forma en que nuestros mapeos traducen a lógica de

primer orden las expresiones OCL relevantes (invariantes y restricciones de autorización) y ii) las heurísticas implementadas por el resolutor SMT.

9.4.1 Motivación

La ingeniería dirigida por modelos (MDE, de sus siglas en inglés) soporta el desarrollo de sistemas de software complejos para la generación de software a partir modelos. La seguridad dirigida por modelos (MDS, de sus siglas en inglés) [9] es una especialización de este paradigma, donde los diseños de sistemas son modelados junto con sus requisitos de seguridad y las infraestructuras de seguridad son generadas directamente a partir de los modelos. Por supuesto, la calidad del código generado depende de la calidad de los modelos fuente. Si los modelos no especifican de forma apropiada el comportamiento del sistema deseado, cabe esperar que el sistema generado correspondiente tampoco lo haga. *Quod natura non dat, Salmantica non praestat.* Nuestra experiencia muestra que incluso usando lenguajes potentes y de alto nivel, es fácil cometer errores lógicos y omisiones. Por lo tanto, es crítico no sólo que el lenguaje de modelado tenga una semántica bien definida, sino que también exista un soporte de herramienta para analizar las propiedades modeladas del sistema.

9.4.2 Categorías de propiedades de seguridad

A continuación, explicaremos cómo usar el mapeo de OCL a lógica de primer orden propuesto en [39] para razonar sobre modelos de seguridad. Este mapeo consiste esencialmente en dos componentes relacionados entre sí: i) un mapeo de los modelos de datos y expresiones OCL de tipo Boolean a fórmulas de primer orden, llamado $\text{ocl2fol}_{\text{def}}$, y ii) un mapeo de expresiones OCL de tipo Boolean a fórmulas de primer orden, llamado ocl2fol. La siguiente Definición formaliza la propiedad principal de este mapeo:

Remark 4 *Sea $\mathcal{D}$ un modelo de datos con las invariantes $expr_1, \ldots, expr_n$ y sea expr una expresión de tipo Boolean. Entonces expr evalúa a* `true` *en toda instancia válida de $\mathcal{D}$ si y sólo si*

$$\begin{aligned}&\text{ocl2fol}_{\text{def}}(\mathcal{D})\\ &\quad \cup \bigcup_{i=1}^{n} \text{ocl2fol}_{\text{def}}(expr_i) \cup \bigcup_{i=1}^{n} \{\text{ocl2fol}(expr_i)\}\\ &\quad \cup\ \text{ocl2fol}_{\text{def}}(expr) \cup \{\neg(\text{ocl2fol}(expr))\}\end{aligned}$$

es insatisfacible.

En lo que sigue, dado un modelo de seguridad $\mathcal{S}$ usamos el término *escenario* para referirnos a cualquier instancia válida del modelo de datos subyacente de $\mathcal{S}$, en el cual un usuario solicita permiso para realizar una acción. Para simplificar, asumiremos que ni el usuario que solicita el permiso ni el recurso sobre el que la acción será ejecutada pueden ser *indefinidos*.

En [43] definimos cuatro categorías o bloques de propiedades de seguridad para el razonamiento sobre políticas FGAC. Cada uno de estos bloques viene acompañado en [43] con una serie de ejemplos ilustrativos. A continuación, se definen estos cuatro bloques de propiedades de seguridad.

En el primer bloque estamos interesados en saber si existe un escenario en el que alguien con rol r pueda ejecutar una acción act. Según la Definición 4, la respuesta será

'No' si y sólo si el siguiente conjunto de fórmulas es insatisfacible:

$$\begin{aligned}&\text{ocl2fol}_{\text{def}}(\mathcal{D}) \cup \ \{\exists(\mathit{caller})\exists(\mathit{self})\exists(\mathit{target})\exists(\mathit{value})\\ &(\text{ocl2fol}(\mathit{caller}.\mathsf{role} = \mathsf{r}) \wedge \text{ocl2fol}(\text{Auth}(\mathcal{S}, r, \mathit{act})))\}.\end{aligned}$$

En el segundo bloque estamos interesados en saber si existe un escenario en el que alguien con rol r no pueda ejecutar una acción act. Según la Definición 4, la respuesta será 'No' si y sólo si el siguiente conjunto de fórmula es insatisfacible:

$$\begin{aligned}&\text{ocl2fol}_{\text{def}}(\mathcal{D}) \cup \{\exists(\mathit{caller})\exists(\mathit{self})\exists(\mathit{target})\exists(\mathit{value})\\ &(\text{ocl2fol}(\mathit{caller}.\mathsf{role} = r) \wedge \neg(\text{Auth}(\mathcal{S}, r, \mathit{act})))\}.\end{aligned}$$

En el tercer bloque estamos interesados en saber si existe un escenario en el que nadie con rol r pueda ejecutar una acción act. Según la Definición 4, la respuesta será 'No' si y sólo si el siguiente conjunto de fórmulas es insatisfacible:

$$\begin{aligned}&\text{ocl2fol}_{\text{def}}(\mathcal{D}) \cup \{\exists(\mathit{self})\exists(\mathit{target})\exists(\mathit{value})\forall(\mathit{caller})\\ &\quad(\text{ocl2fol}(\mathit{caller}.\mathsf{role} = r) \Rightarrow\\ &\qquad\neg(\text{ocl2fol}(\text{Auth}(\mathcal{S}, r, \mathit{act}))))\}.\end{aligned}$$

En el cuarto bloque estamos interesados en saber si para todo escenario, existe al menos un objeto sobre el que nadie con rol r pueda ejecutar una acción act. Según la Definición 4, la respuesta es 'Sí' si y sólo si el siguiente conjunto de fórmulas es insatisfacible:

$$\begin{aligned}&\text{ocl2fol}_{\text{def}}(\mathcal{D}) \cup \{\forall(\mathit{self})\exists(\mathit{target})\exists(\mathit{value})\exists(\mathit{caller})\\ &(\text{ocl2fol}(\mathit{caller}.\mathsf{role} = r) \wedge \text{ocl2fol}(\text{Auth}(\mathcal{S}, r, \mathit{act})))\}.\end{aligned}$$

9.4.3 Pruebas de propiedades de seguridad

Presentamos brevemente nuestra experiencia en el uso del resolutor SMT Z3 [45] para obtener automáticamente las respuestas a los ejemplos descritos en [43], pertenecientes a cada una de los bloques o categorías de propiedades de seguridad descritos arriba. En [43] mostramos, para cada ejemplo, el tiempo que Z3 tarda en dar una respuesta (el cual es siempre menor de un segundo), la respuesta que devuelve (la cual es siempre la esperada) y cuando la respuesta es *sat*, el modelo de primer orden que Z3 genera cuando el conjunto de fórmulas de entrada es satisfacible. Las fórmulas de primer orden que sirven como entrada para Z3 han sido generadas mediante nuestra herramienta SecProver [79]. Esta herramienta toma los siguientes parámetros: un modelo de datos, un modelo de seguridad, un conjunto (posiblemente vacío) de invariantes sobre el modelo de datos, una acción, un rol, un conjunto (posiblemente vacío) de restricciones adicionales y un *tipo de pregunta*[4]. SecProver genera de forma automática un conjunto de fórmulas de primer orden cuya satisfacibilidad determinará, de acuerdo con nuestra metodología, la respuesta a la pregunta dada.

[4]El tipo de pregunta hace referencia al bloque o categoría de propiedad de seguridad a comprobar.

Capítulo 10

Validación del desarrollo dirigido por modelos

Para proporcionar evidencias de la usabilidad de nuestra metodología, en este Capítulo hacemos una breve descripción de cinco aplicaciones de gestión segura de datos desarrolladas mediante el uso de ActionGUI, centrándonos en la aplicación eHRMApp. Dicha aplicación está basada en la gestión de salud electrónica (eHealth) propuesto en NESSoS, la Red de Excelencia sobre la creación de Servicios y Sistemas de Software Seguros para el Internet del Futuro [71].

En general, nuestra experiencia con este caso de estudio demuestra el potencial de la metodología para el desarrollo de aplicaciones reales. Primero, el uso de una transformación de modelos y de un generador de código libera al desarrollador de programar restricciones de autorización de grano fino y de insertarlas con los argumentos correctos, en todos los lugares requeridos a lo largo del código de la aplicación. Excepto para aplicaciones pequeñas, esta tarea es tediosa y propensa a errores ya que el número de acciones de datos asociadas a los eventos puede ser del orden de cientos. Segundo, nuestra metodología soporta modularidad y separación de tareas. En particular, el modelo de seguridad puede ser modificado de forma independiente al modelo de GUI sin la necesidad de reprogramar ni reinsertar las nuevas restricciones de autorización de grano fino, ya que esto se hace de forma automática por nuestra transformación de modelos.

10.1 Una aplicación segura de eHealth

La aplicación eHRMApp está basada en un caso de estudio propuesto en NESSoS, la Red de Excelencia sobre la creación de Servicios y Sistemas de Software Seguros para el Internet del Futuro [71]. El caso de estudio de eHealth consiste en un sistema web para la gestión de registros de salud electrónicos (EHRM). Los registros de salud electrónicos (EHR) almacenan información creada por, o en nombre de, un profesional de la salud en el contexto del cuidado de un paciente.

Los registros de salud electrónicos son altamente sensibles y, por lo tanto, su acceso debe ser controlado. Parte del reto en este caso de estudio fue modelar la política de control de acceso y construir una aplicación que la aplique en tiempo de ejecución. La política consiste en varias reglas de autorización similares a: *El criterio de control de acceso a un EHR depende, además de otros, del tipo de EHR. Por ejemplo, un registro altamente sensible de un paciente debería estar disponible sólo para el doctor que trata a dicho paciente (y quizás algunos otros, en ocasiones excepcionales).* Tales reglas necesitan control de acceso de grano fino donde las decisiones de control de acceso dependen, no

sólo de las credenciales del usuario, sino también, de la satisfacción de restricciones en el estado de la capa persistente, es decir, en los valores de los elementos almacenados.

En general, el caso de estudio de eHealth es interesante como un ejemplo de desarrollo de una aplicación de gestión segura de datos y proporciona una prueba de concepto de la aplicación de la metodología ActionGUI sobre un problema de relevancia industrial. A continuación describimos brevemente las principales características de la aplicación eHRMApp.

El modelo de datos El modelo de datos completo de la aplicación contiene 18 entidades, 40 atributos y 48 extremos de asociación.

Las invariantes del modelo de datos El modelo de datos completo de la aplicación está restringido por 66 invariantes formalizadas en OCL.

El modelo de seguridad Los registros de salud electrónicos son, por su naturaleza, altamente sensibles. Por ello, el caso de estudio de NESSoS define informalmente la política que regula el acceso a estos registros. Como es de esperar, la autorización para llevar a cabo ciertas acciones no está sólo basada en los roles sino también en el contexto. En otras palabras, la política de control de acceso de eHRMApp es de grano fino.

El modelo completo de seguridad de la aplicación contiene 5 roles y 573 permisos, donde cada permiso autoriza a los usuarios con un rol a ejecutar una acción bajo la satisfacción de una restricción de autorización formalizada en OCL.

Los modelos de seguridad de ActionGUI son formulados en términos de los datos de la aplicación. Esta formalización es independiente de cómo los datos son visualizados o accedidos desde la interfaz gráfica de usuario de la aplicación.

El modelo de GUI El modelo completo de GUI de la aplicación contiene 8 ventanas para los siguientes casos de uso: acceso la información de un centro médico, registro de un nuevo paciente, revisión de la información de un paciente, reasignación de un paciente a un doctor y departamento, acceso a las opciones reservadas al director del centro médico, registro de un profesional en el sistema y reasignación de un profesional a un departamento. Éstos son algunos datos sobre el tamaño del modelo de GUI: i) widgets: 19 botones, 73 etiquetas, 19 cajas de texto, 5 campos booleanos, 1 campo de fecha, 1 desplegable y 9 tablas; ii) sentencias: 34 sentencias if-then-else; iii) acciones sobre datos: 11 acciones de crear, 41 acciones de actualizar (update), 5 acciones de crear un enlace y 2 acciones de eliminar un enlace; iv) acciones de GUI: 157 acciones set y 7 acciones de abrir (open) y v) expresiones OCL: 361 expresiones (77 no literales).

El modelo de GUI con seguridad Tal y como se explica en la Sección 8.3, el núcleo de ActionGUI es la función de transformación de modelos que, esencialmente, prefija cada acción sobre datos en el modelo de GUI con una comprobación de autorización especificada en el modelo de seguridad. El modelo completo de GUI contiene 59 acciones sobre datos y, por lo tanto, el modelo de GUI con seguridad de la aplicación automáticamente generado contiene el mismo número de comprobaciones de autorización.

La generación de la aplicación El conjunto de herramientas ActionGUI genera la aplicación completa en menos de 10 segundos. El archivo .war generado incluye la librería de Vaadin así como otras librerías externas. La librería de Vaadin es responsable del 70 % del tamaño del archivo generado y sólo el 10 % de este archivo corresponde al código que

ActionGUI genera para interpretar los modelos de la aplicación. El tamaño del archivo .war que contiene la aplicación completa es de aproximadamente 12 MB.

10.2 Otras aplicaciones

Aquí presentamos otras cuatro aplicaciones web que hemos desarrollado utilizando ActionGUI. Comenzamos haciendo una breve descripción de estas aplicaciones, a continuación detallamos las diferentes métricas que usamos para medir el tamaño de las aplicaciones desarrolladas y por último concluimos con algunas consideraciones subjetivas.

Gestión de Relaciones entre Clientes (CRMApp) Es una aplicación web para la gestión de clientes de un centro hospitalario. Esta aplicación permite gestionar la información de contactos, incluyendo filtros de contactos basados en diferentes criterios y exportación de los resultados a hojas de cálculo. Ya que los datos de un cliente son altamente sensibles, los datos están sujetos a políticas de control de acceso restrictivas. Por ejemplo, el personal de marketing sólo puede acceder a la información de contacto de aquellos clientes previamente seleccionados como objetivos para una campaña de marketing, a la cual el personal de marketing es asignado. La aplicación también permite un Gestor General para crear campañas de marketing, seleccionar los pacientes objetivos para las campañas y asignar personal de marketing a las campañas.

Gestión del Voluntariado (VMApp) Es una aplicación web para la gestión del programa de voluntarios de un centro de cuidados paliativos. Mediante el uso de esta aplicación, los coordinadores del programa pueden realizar acciones tales como: introducir nuevos voluntarios en el programa, crear, modificar y eliminar tareas de voluntariado y proponer estas tareas a los voluntarios basándose en la disponibilidad temporal y las preferencias de los propios voluntarios. La política de control de acceso estipula, por ejemplo, que los voluntarios están autorizados a modificar únicamente su propia información personal, como sus preferencias y su disponibilidad temporal, y de aceptar o rechazar sus tareas asignadas.

Gestión de Servicio de Comidas (MSMApp) Es una aplicación web para la gestión del servicio de comidas para residencias de estudiantes. Mediante el uso de esta aplicación, un residente puede notificar a la administración de la residencia si va a comer en la cafetería de la residencia, en qué opción de comida dentro de las disponibles y si traerá algún invitado. Un residente debe poder modificar sólo sus propias opciones de comida y siempre dentro de una ventana temporal la cual depende de la opción de comida. Los administradores de la aplicación pueden crear nuevas altas de cuentas de residentes y listar las opciones de comidas solicitadas para cada tipo de comida.

Salas de Conversación (ChatApp) Es una aplicación web basada en salas de conversación. Dicha aplicación proporciona un lugar online de discusión donde los usuarios pueden conversar mediante el envío de mensajes. Los usuarios pueden estar o no registrados. Además de enviar mensajes a una sala de conversación seleccionada, los usuarios también pueden crear y eliminar salas de conversación bajo condiciones específicas.

CRMApp, VMApp y MSMApp son aplicaciones *comerciales*. Han sido desarrolladas para clientes reales y actualmente están siendo usadas por dichos clientes. En contraste, eHRMApp ha sido desarrollada, tal y como explicamos en la Sección 10.1, como parte de un caso de estudio propuesto por socios empresariales en un proyecto europeo. El lector

interesado puede encontrar más información sobre este caso de estudio, así como una versión de demostración de la aplicación eHRMApp, en [1].

Para medir estas aplicaciones nos basamos en dos parámetros: el tamaño de los datos de la aplicación y el tamaño de los modelos subyacentes de la aplicación. Con respecto al tamaño de los datos de la aplicación tenemos en cuenta: el número de usuarios registrados, el tamaño actual de la base de datos, el número de tablas definidas en el esquema de la base de datos y el máximo número de filas que actualmente contiene cada una de las tablas. Con respecto al tamaño de los modelos subyacentes de la aplicación, tenemos en cuenta: el tamaño de los archivos XML de cada uno de los modelos, el número de widgets de cada tipo de la aplicación, el número de cada tipo de acción de datos y de acción de GUI asociadas a los eventos de los widgets y el número de expresiones OCL usadas para definir las condiciones en las sentencias de tipo if-then-else, las colecciones fuente en las sentencias iteradoras y los argumentos de las acciones de datos y de GUI. Además, indicamos también cuántas de estas expresiones OCL no son valores literales, el tamaño medio de las expresiones y el número medio de variables de widget que referencian.

Con respecto a la generación de código, el conjunto de herramientas ActionGUI genera de forma automática todas nuestras aplicaciones en menos de un minuto. El archivo .war generado incluye la dependencia a la librería Vaadin y a otras librerías externas (además de, en el caso de las aplicaciones CRMApp y VMApp, código personalizado para enviar correos electrónicos y exportar datos). La dependencia a la librería Vaadin es de hecho responsable del 70 % del tamaño del archivo generado, mientras que sólo un 10 % corresponde al código que ActionGUI genera de manera automática para interpretar los modelos de la aplicación (como es de esperar, el tamaño de este *intérprete* no varía mucho entre aplicaciones). Para nuestros ejemplos, el tamaño medio del archivo .war que contiene cada aplicación es de aproximadamente 12 MB.

10.3 Evaluación

Concluimos este Capítulo con varias consideraciones subjetivas. Ya que dichas consideraciones están basadas en nuestra propia experiencia, su importancia no debe ser exagerada. Sin embargo, pueden dar al lector una impresión de lo que implica el uso de ActionGUI.

Primero, la curva de aprendizaje de ActionGUI es moderada. El tiempo requerido para aprender a modelar una aplicación de gestión segura de datos depende de la familiarización que el modelador tiene con los diagramas de clases, diagramas de seguridad y, principalmente, con OCL. Asumiendo un modesto conocimiento de UML, el aprendizaje requerido para modelar aplicaciones (reales) mediante el uso de ActionGUI lleva menos de 4 horas, que es discutiblemente mucho menos tiempo del que se requiere para adquirir el suficiente conocimiento de tecnologías web para poder programar (y desplegar de forma correcta) estas aplicaciones. Esta estimación está basada, en parte, en nuestra experiencia impartiendo el desarrollo dirigido por modelos de aplicaciones de gestión segura de datos a estudiantes en el ETH de Zúrich, los cuales tuvieron que modelar proyectos mediante el uso de ActionGUI.

Segundo, el tiempo que se necesita para modelar una aplicación de gestión segura de datos depende de la experiencia del modelador (en particular, de su destreza en el uso del lenguaje OCL) y de la complejidad de la aplicación (en particular, del número y el tamaño de las expresiones OCL que se usan en el modelo). Sin embargo, en general, el modelado de una ventana de *menú*, que contiene 6 botones que al pinchar sobre ellos se abren otras ventanas, puede llevar menos de 30 minutos. En contraste, el modelado de una

ventana que contiene un *formulario online*, con 10 campos de texto y un botón que cuando pinchas sobre él primero comprobará que las entradas han sido correctamente rellenadas y luego enviará en formulario (es decir, actualizará la base de datos), puede llevar 1 o 2 horas. Y aproximadamente el mismo tiempo es requerido para modelar una ventana del tipo *seleccionar y mostrar*, que contiene una tabla con 5 columnas y un desplegable y que la información mostrada en la tabla cambiará dependiendo de la selección del desplegable. La conclusión es que el tiempo de modelado es directamente proporcional al número y el tamaño de las expresiones OCL usadas en los modelos. Esto no es una sorpresa ya que estas expresiones son las que definen la lógica de la aplicación.

Finalmente, las principales ventajas de nuestra metodología tienen que ver con la calidad y la facilidad de mantenimiento de las aplicaciones generadas, las cuales se generan de forma automática mediante el uso de nuestra transformación de modelos y generador de código. Primero, esta transformación libera de manera efectiva al desarrollador de tener que programar restricciones de autorización de grano fino y de insertarlas con los argumentos correctos, en todos los puntos del código de la aplicación requeridos. Salvo para aplicaciones pequeñas, esta tarea es tediosa y propensa a errores ya que el número de acciones de datos asociadas a eventos puede ser del orden de cientos como, por ejemplo, las aplicaciones CRMApp y VMApp. Además, estas acciones de datos son normalmente llamadas con diferentes argumentos cada vez. Segundo, nuestra metodología soporta modularidad y separación de tareas. En particular, el modelo de seguridad se puede modificar de manera independiente al modelo de GUI sin tener que preocuparnos de reprogramar ni reinsertar todas las nuevas restricciones de autorización de grano fino, ya que esto se realiza de forma automática por nuestra transformación de modelos. De nuevo, debido al número de estas comprobaciones junto con la complejidad de las correspondientes autorizaciones, argumentamos que, para aplicaciones reales de gestión segura de datos nuestra metodología reduce de forma efectiva los costes de mantenimiento.

Capítulo 11

Trabajo relacionado

En este Capítulo, nos centramos en trabajos que están directamente relacionados con la principal contribución de nuestro trabajo, a saber, ActionGUI como una metodología dirigida por modelos y con soporte de herramienta para el desarrollo de aplicaciones de gestión segura de datos. Hemos organizado el trabajo relacionado en dos secciones no completamente disjuntas. Primero, compararemos ActionGUI con otras propuestas de modelado de aplicaciones de gestión segura de datos y, en particular, con sus interfaces gráficas de usuario. Luego, compararemos ActionGUI con otras herramientas (en este caso, herramientas comerciales) para el desarrollo de aplicaciones de gestión segura de datos.

Modelado de aplicaciones de gestión segura de datos Como herramienta de modelado, UWE [14, 21, 19, 63] proporciona al modelador un nivel de abstracción mayor que ActionGUI. En particular, las acciones ejecutadas por los eventos de los widgets son descritas en UWE mediante el uso del lenguaje natural. De este modo, salvo que los modelos sean refinados de forma apropiada tal y como se argumenta en [63], UWE no soporta generación de código. Por contra, UWE proporciona diagramas específicos para el modelado de *presentaciones* y *navegaciones* de GUI, los cuales facilitan la tarea de modelar interfaces gráficas. En este sentido, en [20] definimos un mapeo que transforma modelos UWE de alto nivel a modelos de ActionGUI más concretos que, una vez completados por el modelador, pueden ser directamente usados para generar las aplicaciones deseadas. Finalmente, [19] extiende UWE con el uso de SecureUML para el modelado de políticas de seguridad. Sin embargo, este trabajo no usa una transformación de modelos para elevar la política de seguridad al nivel de la GUI. En su lugar, el modelador de UWE es el responsable de añadir todas las comprobaciones de autorización apropiadas en el modelo de GUI.

Como ActionGUI, ZOOM [60] permite a los modeladores de GUIs especificar widgets, sus eventos y sus acciones. Además, mediante el uso de una extensión de Z [86] se pueden especificar las condiciones de las acciones y sus argumentos de forma similar a como ocurre en ActionGUI mediante el uso de OCL. A diferencia de ActionGUI, ZOOM no proporciona un lenguaje para el modelado de la seguridad ya que aspectos relacionados con seguridad no se consideran de forma explícita en esta metodología. Además, ZOOM no soporta generación de código. Sólo proporciona intérpretes para la animación de modelos.

En contraste con ActionGUI, UWE y ZOOM, las metodologías presentadas en [37, 38, 51] no proporcionan un lenguaje para el modelado de GUIs. En su lugar, implementan diferentes *reglas* para derivar de forma automática las GUIs basadas en el modelo de datos de la aplicación (como en [37, 38]) o en escenarios prototípicos de la aplicación (como en [51]). Como es de esperar, el comportamiento de las GUIs obtenidas es limitado y, basándonos en nuestra experiencia, insuficiente para hacer frente a la lógica contenida en

aplicaciones reales de gestión de datos. Además, aspectos de seguridad no son abordados por estas propuestas.

Hay otro trabajo relacionado que se encuentra entre los dos extremos de modelado completo de GUIs y derivación completa de GUIs. La propuesta OO-method [74, 69] soporta la construcción de GUIs mediante el uso de *patrones* de interfaces gráficas. Estos patrones especifican las posibles interacciones con los datos de la aplicación basados en las clases, atributos y asociaciones que están declaradas en el modelo de datos subyacente. Esta metodología tiene la ventaja de reducir el tiempo requerido para el modelado de GUIs. Sin embargo, los patrones de interfaces gráficas imponen restricciones en el tipo de GUIs que se pueden modelar, tanto en su estructura como en su comportamiento. Además, esta metodología soporta control de acceso basado en roles pero no soporta control de acceso de grano fino. El Lenguaje de Modelado de Interacción de Flujo (IFML, de sus siglas en inglés) [85] es otra metodología que se encuentra en esta categoría. IFML es el lenguaje de modelado estándar de la OMG para expresar contenido, interacciones de usuario y control del comportamiento de los *front-ends* de las aplicaciones, así como el enlace de las capas de persistencia y de lógica de negocio de las aplicaciones. IFML es una versión mejorada de su predecesor, el lenguaje WebML [28]. Concretamente, WebML estaba orientado a al diseño basado en componentes de modelos específicos de plataforma (PSMs, de sus siglas en inglés) para *front-ends* de aplicaciones web; mientras que IFML está orientado al diseo basado en componentes de modelos independientes de plataforma (PIMs, de sus siglas en inglés) para *front-ends* de cualquier tipo de aplicación. IFML está integrado en el conjunto de herramientas WebRatio [84]. Su lenguaje de diseño basado en componentes, de nuevo, restringe el tipo de GUIs que pueden ser modeladas. Además, aspectos de seguridad no son del todo considerados en IFML.

Otras metodologías que se encuentran en esta categoría son [78, 64]. En ambos casos, el modelador tiene que asociar a cada contenedor de widgets el tipo de dato al que se accede mediante su uso. Como antes, las posibles interacciones con el modelo de datos subyacente están limitadas por el comportamiento implementado por defecto para estos contenedores de widgets. Aspectos de seguridad no son considerados tampoco en esta metodología.

Finalmente, existen metodologías centradas principalmente es el soporte de diseño de interfaces gráficas a distintos niveles de abstracción. Ejemplos importantes son el Lenguaje XML de Interfaces de Usuario (XUL, de sus siglas en inglés) [70] y el Lenguaje de Marcado Extensible de Interfaces de Usuario (UsiXML, de sus siglas en inglés) [65]. XUL es el lenguaje basado en XML de Mozilla para la construcción de interfaces de usuario para aplicaciones de la plataforma Mozilla (como Firefox). UsiXML en un lenguaje de marcado basado en XML que describe interfaces gráficas para múltiples contextos de uso tales como Interfaces de Caracteres de Usuario (CUIs), Interfaces Gráficas de Usuario (GUIs), Interfaces de Auditoría de Usuario e Interfaces Multimodales de Usuario.

Claramente, ActionGUI está diseñado para un propósito distinto al de XUL y UsiXML. En particular, ActionGUI está diseñado para el desarrollo de aplicaciones de gestión *segura* de datos. Una decisión de diseño clave en ActionGUI fue asegurar que el modelo de seguridad y el modelo de GUI "hablen" el mismo lenguaje. Según nuestros conocimientos, ni XUL ni UsiXML tienen en cuenta los aspectos de seguridad de las interfaces gráficas. Además, ActionGUI está diseñado para el desarrollo *dirigido por modelos* de aplicaciones de gestión segura de datos y esto tiene dos claras consecuencias. Primero, los lenguajes de modelado de ActionGUI están diseñados para ser tecnológicamente agnósticos, en contraste con XUL, el cual está fuertemente ligado a la plataforma de Mozilla. Segundo, los lenguajes de modelado de ActionGUI están diseñados para soportar la generación automática, a partir de los modelos, de aplicaciones listas para ser desplegadas. Como

consecuencia, los modelos de ActionGUI son más concretos que los modelos generales de UsiXML, los cuales pueden ser definidos en cualquiera de los cuatro niveles de abstracción especificados en la Entorno de Referencia Cameleon (CRF) [25]. En ActionGUI en particular, un modelador de GUI trabaja siempre al nivel Concreto de UI de CRF, mientras que el archivo WAR (archivo de aplicación web) generado a partir del modelo de GUI (junto con los modelos de datos y seguridad asociados) pertenece al nivel Final de UI de CRF.

Finalmente, mencionar que [83] ha llevado a cabo un prometedor trabajo de extensión de nuestra metodología para trabajar con procesos de negocio, los cuales son típicamente definidos en el nivel de abstracción de Tareas y Conceptos de CRF. En esta línea, sería interesante investigar formas de extender nuestra metodología para soportar el modelado de UIs al nivel Abstracto de UI de CRF, donde los detalles de interacción son abstraídos.

Desarrollo de aplicaciones de gestión segura de datos Existen también otras herramientas como WebRatio [84], Olivanova [26] y Lightswitch [68] que soportan metodologías de desarrollo para la construcción de aplicaciones de gestión de datos, similares a la metodología de ActionGUI. En estas herramientas, el desarrollo de la aplicación comienza con la construcción de un modelo de datos que refleja la estructura de datos requerida por la base de datos. El proceso de desarrollo continua con la aplicación sobre el modelo de datos de diferentes patrones de generación de interfaces gráficas. Estos patrones permiten la recuperación de datos, la edición de datos, la creación de datos y la búsqueda en la base de datos. En contraste con lo que estas herramientas pueden proporcionar, el conjunto de herramientas ActionGUI ofrece a los desarrolladores flexibilidad total para crear los diseños sin las restricciones impuestas por la obligatoriedad de usar un número fijo de patrones dados. Las herramientas presentadas arriba imponen además una mayor restricción a nivel de gestión de datos, es decir, al nivel de acceso y visualización de datos: la información que puede ser referenciada, y por lo tanto que puede ser accedida y visualizada en una pantalla, puede únicamente venir de una tabla de la base de datos, o a lo sumo, de dos tablas que están relacionadas entre ellas mediante una navegación directa (de un único paso).

Estas tres herramientas, WebRatio, Olivanova y Lightswitch soportan la definición y generación de políticas RBAC a diferentes niveles de granularidad. Lightswitch soporta la concesión o denegación de permisos de usuario, para los diferentes roles (cuyos comportamientos reales tienen que ser programados de forma manual), de ejecución de acciones sobre entidades de crear, leer, modificar y borrar. WebRatio y Olivanova soportan también la concesión y denegación de permisos de usuario, para los diferentes roles, de ejecución de conjuntos de acciones similares sobre propiedades de entidades, individualmente. En WebRatio y Olivanova el rol de las restricciones de autorización puede seguir las reglas de las precondiciones sujetas a la invocación de acciones a través de una interfaz gráfica concreta. Sin embargo, ninguna de estas herramientas implementa un algoritmo capaz de trasladar a las interfaces gráficas, la política de seguridad que gobierna el acceso a los datos.

Capítulo 12

Conclusiones y trabajo futuro

La metodología que proponemos constituye un extenso desarrollo de la idea de seguridad dirigida por modelos(MDS, de sus siglas en inglés) [9]. Las dos principales innovaciones son un lenguaje expresivo para modelar la interfaz gráfica de usuario y el comportamiento de una aplicación y una transformación de modelos que traslada la política de seguridad, especificada sobre el modelo de datos de la aplicación, al modelo de comportamiento de GUI. Nuestra función de transformación captura la idea de que las políticas de autorización, que regulan las complejas transacciones, pueden ser generadas uniformemente a partir de políticas más simples sobre datos. A pesar de nuestro uso de lenguajes expresivos de modelado hemos mostrado para aplicaciones de gestión de datos, que es posible generar de manera automática aplicaciones completas y listas para desplegar. Nuestra metodología está soportada por el conjunto de herramientas ActionGUI. Las aplicaciones presentadas muestran el potencial del conjunto de herramientas ActionGUI, para el desarrollo de aplicaciones del mundo real.

En el futuro, planeamos extender el desarrollo del proyecto ActionGUI en dos direcciones principales:

Mejora del conjunto de herramientas ActionGUI. Todavía hay mucho trabajo por delante para convertir este conjunto de herramientas en una plataforma de desarrollo completa, robusta y de nivel industrial. A corto plazo, planeamos desarrollar mejores editores de modelos y un mejor soporte para integrar código personalizado. Luego a medio plazo, también nos gustaría soportar el análisis de modelos basado en la semántica formal de nuestros modelos y en la corrección de nuestra transformación de modelos. A continuación, mostramos ejemplos de preguntas que nos gustaría poder contestar de manera formal. ¿Preservarán las invariantes del modelo de datos todas las secuencias de acciones ejecutadas por cada uno de los eventos en el modelo? ¿Forzarán alguna vez las comprobaciones de autorización un roll-back de una transacción? ¿Son redundantes las condiciones en el modelo de GUI con respecto a las comprobaciones de autorización generadas por la transformación de modelos? El soporte de análisis nos ayudaría a optimizar el código generado y soportaría actividades seguras como la certificación de sistemas. Finalmente, a largo plazo, nos gustaría soportar la generación de GUIs para diferentes plataformas como dispositivos móviles. También planeamos añadir soporte para manejar *políticas de privacidad*: modelado y generación de código para forzar que el uso de los datos siga el propósito para el que los datos fueron obtenidos y puedan implicar obligaciones.

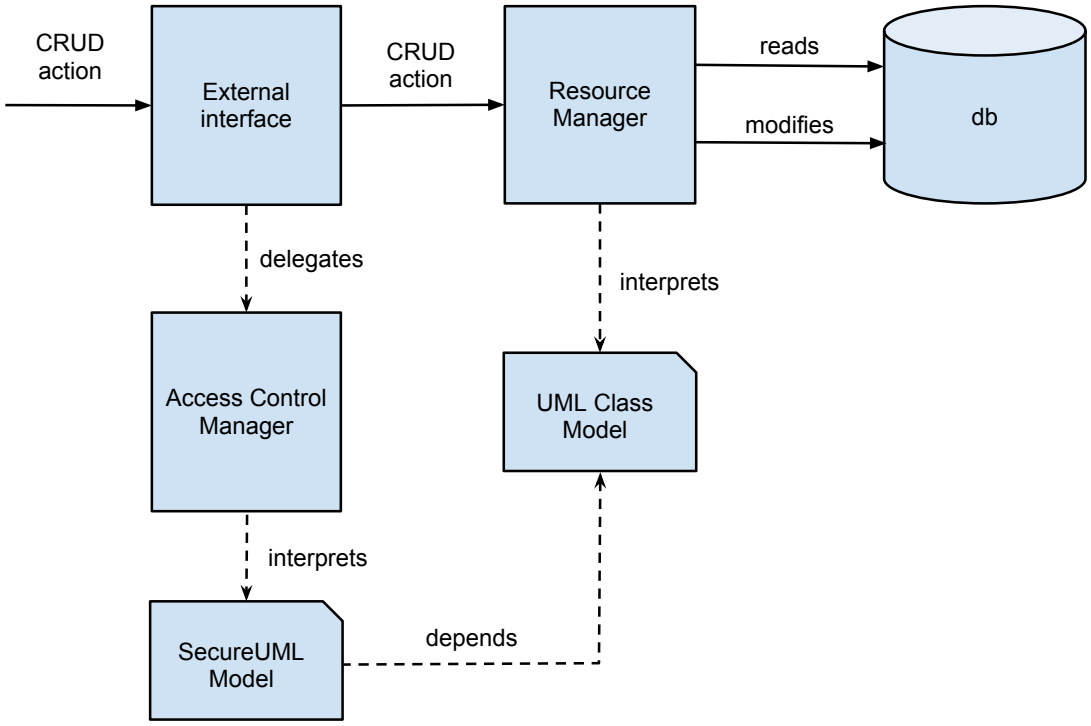


Figura 12.1: Arquitectura Actual de SecureDAO

Extensión de la aplicabilidad de la metodología y herramientas de ActionGUI. SecureDAO es una librería Java desarrollada por Gonzalo Ortiz de Jaureguizar para forzar el control de acceso de grano fino en las aplicaciones ActionGUI. Está diseñado para ejecutar cualquiera de las llamadas acciones CRUD (crear, leer, modificar y borrar datos) pero sólo si la restricción de autorización especificada por esta acción se satisface en el modelo de seguridad dado. Las acciones CRUD son ejecutadas por el *gestor de recursos* de SecureDAO, mientras que las restricciones de autorización son comprobadas por el *gestor de control de acceso* de SecureDAO. SecureDAO posee una arquitectura muy modularizada, la cual se ilustra en la Figura 12.1.

En el futuro, planeamos desarrollar un mecanismo de aplicación de seguridad orientado a empresas y dirigido por modelos, extendiendo SegureDAO en las siguientes líneas:

- *SecureDAO como servicio*, de manera que no sólo una aplicación sino múltiples aplicaciones puedan gestionar los mismos recursos con las mismas políticas de control de acceso.

- Un *gestor de recursos más genérico*, de manera que no sólo datos sino también métodos puedan ser gestionados.

- Un *gestor de control de acceso más genérico*, de manera que no sólo políticas basadas en SecureUML sino también otras políticas puedan ser comprobadas de manera efectiva.

Para cumplir estos objetivos necesitaremos abordar los siguientes retos tecnológicos y de investigación:

- SecureDAO está actualmente implementado como una librería Java. Para proporcionar SecureDAO como un servicio necesitamos encapsularlo en un aplicación empresarial, que pueda ser usada por diferentes aplicaciones de clientes a través de puntos de entrada a nivel de aplicación. Nuestro reto aquí es proveer de manera automática puntos de entrada de SecureDAO, fáciles de usar y sin errores de tipos, basados en el modelo de datos subyacente de la aplicación del cliente.

- La aplicabilidad del gestor de recursos de SecureDAO está actualmente limitada por los datos que son almacenados en las bases de datos MySQL, los cuales usan esquemas de bases de datos muy específicos. Un gestor de recursos más general debería, lo primero de todo, ser capaz de gestionar datos almacenados en diferentes (incluso no relacionales) bases de datos. Nuestro reto aquí es construir un gestor de recursos genérico, el cual pueda ser instanciado proporcionando un mapeo bien definido del modelo de datos subyacente de la aplicación del cliente al esquema de la base de datos de la aplicación del cliente. Además, un gestor de recursos más general debería permitir ejecutar también acciones no CRUD (a saber, métodos generales). Nuestro reto aquí es proporcionar una conexión entre el gestor de recursos y las librerías que implementan las acciones no CRUD requeridas.

- La aplicabilidad del gestor de control de acceso de SecureDAO está actualmente limitada a políticas declaradas mediante el uso de modelos SecureUML, cuyas restricciones de autorización están formalizadas mediante el uso del Lenguaje de Restricción de Objetos (OCL, de sus siglas en inglés) [73]. Nuestro reto aquí es construir un gestor genérico de control de acceso, el cual pudiera ser instanciado proporcionando una semántica de evaluación bien definida de un lenguaje de autorización arbitrario.

Bibliography

[1] ActionGUI. The ActionGUI project, 2014. `http://www.actiongui.org`.

[2] T. Baar and S. Markovic. The RoclET tool. `http://www.roclet.org/index.php`, 2007.

[3] A. Baroni and F. B. e Abreu. An OCL-based formalization of the MOOSE metric suite. In *ECOOP Workshop on Quantitative Approaches in Object Oriented Software Engineering*, Darmstadt, Germany, July 2003.

[4] A. L. Baroni and F. B. e Abreu. A Formal Library for Aiding Metrics Extraction. In *International Workshop on Object-Oriented Re-Engineering at ECOOP'2003*, Darmstadt, Germany, July 2003.

[5] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.

[6] D. Basin, M. Clavel, M. A. G. de Dios, and C. Dania. ActionGUI semantics. Technical report, IMDEA & ETH, 2013. `http://www.actiongui.org`.

[7] D. Basin, M. Clavel, J. Doser, and M. Egea. A metamodel-based approach for analyzing security-design models. In G. Engels, B. Opdyke, D. Schmidt, and F. Weil, editors, *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MODELS '07)*, volume 4735 of *LNCS*, pages 420–435. Springer-Verlag, 2007.

[8] D. Basin, M. Clavel, J. Doser, and M. Egea. Automated analysis of security-design models. *Information and Software Technology*, 51(5):815–831, 2009.

[9] D. Basin, M. Clavel, and M. Egea. A decade of model-driven security. In *Proceedings of the 16th ACM symposium on Access control models and technologies (SACMAT 2011)*, volume 1998443, pages 1–10, Innsbruck, Austria, 2011. New York, NY, USA.

[10] D. Basin, M. Clavel, M. Egea, M. A. G. de Dios, C. Dania, G. Ortiz, and J. Valdazo. Model-driven development of security-aware guis for data-centric applications. In *Foundations of Security Analysis and Design VI - FOSAD Tutorial Lectures*, volume 6858 of *LNCS*, pages 101–124. Springer, 2011.

[11] D. Basin, J. Doser, and T. Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology*, 15(1):39–91, 2006.

[12] D. A. Basin, M. Clavel, M. Egea, M. A. G. de Dios, and C. Dania. A model-driven methodology for developing secure data-management applications. *IEEE Transactions on Software Engineering*, 40(4):324–337, 2014.

[13] D. A. Basin, M. Clavel, M. Egea, and M. Schläpfer. Automatic generation of smart, security-aware GUI models. In F. Massacci, D. S. Wallach, and N. Zannone, editors, *Engineering Secure Software and Systems, Second International Symposium, ESSoS 2010, Pisa, Italy, February 3-4, 2010. Proceedings*, volume 5965 of *LNCS*, pages 201–217. Springer, 2010.

[14] H. Baumeister, N. Koch, and L. Mandel. Towards a UML extension for hypermedia design. In R. B. France and B. Rumpe, editors, *Proc. of UML'99*, LNCS, pages 614–629. Springer, 1999.

[15] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, 2007.

[16] F. Brito e Abreu. Using OCL to formalize object oriented metrics definitions. Technical Report ES007/2001, FCT/UNL and INESC, Portugal, June 2001. `http://www.iro.umontreal.ca/~sahraouh/qaoose01/OCL_Metrics.pdf`.

[17] A. D. Brucker. *An Interactive Proof Environment for Object Oriented Specifications*. PhD thesis, ETH Zurich, 2007.

[18] A. D. Brucker and B. Wolff. The HOL-OCL tool. `http://www.brucker.ch/projects/hol-ocl/index.en.html`, 2007. ETH Zurich.

[19] M. Busch. Integration of security aspects in web engineering. Master's thesis, Institut für Informatik, Ludwig-Maximilians-Universität, München, Germany, 2011.

[20] M. Busch and M. A. G. de Dios. ActionUWE: Transformation of UWE to ActionGUI models, 2012. `http://uwe.pst.ifi.lmu.de/publications/ActionUWE.pdf`.

[21] M. Busch and N. Koch. MagicUWE - a case tool plugin for modeling web applications. In M. Gaedke, M. Grossniklaus, and O. Díaz, editors, *Proc. of ICWE'09*, volume 5648 of *LNCS*, pages 505–508. Springer, 2009.

[22] J. Cabot, R. Clarisó, and D. Riera. A tool for the formal verification of UML/OCL models using constraint programming, 2007. `http://gres.uoc.edu/UMLtoCSP/`.

[23] J. Cabot, R. Clarisó, and D. Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *ASE '07: Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering*, New York, NY, USA, 2007. ACM.

[24] J. Cabot and E. Teniente. Transformation techniques for OCL constraints. *Science Computer Programming*, 68(3):152–168, 2007.

[25] G. Calvary, J. Coutaz, L. Bouillon, M. Florins, Q. Limbourg, L. Marucci, F. Paternò, C. Santoro, N. Souchon, D. Thevenin, and J. Vanderdonckt. The CAMELEON reference framework, 2002. `http://giove.isti.cnr.it/projects/cameleon.html`.

[26] Care Technologies. Olivanova – the programming machine, 2011. `http://www.care-t.com`.

[27] J. Carsí, I. Ramos, A. Boronat, and A. Gómez. The MOMENT: MOdel manageMENT framework project. `http://moment.dsic.upv.es`, 2008.

[28] S. Ceri and P. Fraternali. The web modeling language - WebML, 2003. `http://www.webml.org`.

[29] D. Chiorean, M. Bortes, and D. Corutiu. Proposals for a widespread use of OCL. In *Tool Support for OCL and Related Formalisms - Needs and Trends– MoDELS'05 Conference Workshop*, pages 68–82, 2005.

[30] D. Chiorean, M. Bortes, D. Corutiu, C. Botiza, and A. Carcu. An OCL environment (OCLE) 2.0.4. `http://lci.cs.ubbcluj.ro/ocle/`, 2005. Laboratorul de Cercetare in Informatica, University of BABES-BOLYAI.

[31] M. Clavel and M. Egea. ITP/OCL: A rewriting-based validation tool for UML+OCL static class diagrams. In *AMAST'06: 11th International Conference on Algebraic Methodology and Software Technology*, volume 4019 of *LNCS*, Kuressaare, Estonia, July 2006. Springer-Verlag.

[32] M. Clavel and M. Egea. The ITP/OCL tool. `http://maude.sip.ucm.es/itp/ocl/`, 2008.

[33] M. Clavel, M. Egea, and M. A. G. de Dios. Building an efficient component for ocl evaluation. *ECEASST*, 15, 2008.

[34] M. Clavel, M. Egea, and M. A. G. de Dios. Checking unsatisfiability for ocl constraints. *ECEASST*, 24, 2009.

[35] M. Clavel, M. Egea, and V.Torres. Model metrication in MOVA: A metamodel based approach using ocl. `http://maude.sip.ucm.es/~clavel/pubs/CET07.pdf`, 2007.

[36] M. Clavel, V. Silva, C. Braga, and M. Egea. Model-driven security in practice: An industrial experience. In I. Schieferdecker and A. Hartman, editors, *ECMDA-FA '08: Proceedings of Model Driven Architecture - Industrial Track*, volume 5095 of *LNCS*, pages 327–338, Berlin–Heidelberg, 2008. Springer-Verlag.

[37] A. M. R. da Cruz. *Automatic Generation of User Interfaces from Rigorous Domain and Use Case Models*. PhD thesis, Faculdade de Engenharia da Universidade do Porto, September 2010.

[38] A. M. R. da Cruz and J. P. Faria. A metamodel-based approach for automatic user interface generation. In D. C. Petriu, N. Rouquette, and Ø. Haugen, editors, *Part I of Proc. of Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010.*, LNCS, pages 256–270. Springer, 2010.

[39] C. Dania and M. Clavel. Ocl2fol+: Coping with undefinedness. In J. Cabot, M. Gogolla, I. Ráth, and E. D. Willink, editors, *OCL@MoDELS: Proceedings of the MODELS 2013 OCL Workshop co-located with the 16th International ACM/IEEE Conference on Model Driven Engineering Languages and Systems (MODELS 2013), Miami, USA, September 30, 2013*, volume 1092 of *CEUR Workshop Proceedings*, pages 53–62. CEUR-WS.org, 2013.

[40] Database Systems Group. The UML specification environment (USE) tool, 2006. `http://sourceforge.net/apps/mediawiki/useocl/index.php?title=Main_Page`.

[41] M. A. G. de Dios, M. Clavel, and M. Egea. The Eye OCL Software (EOS), 2008. `http://maude.sip.ucm.es/eos`.

[42] M. A. G. de Dios, C. Dania, D. Basin, and M. Clavel. Model-driven development of a secure eHealth application. In *Engineering Secure Future Internet Services*, volume 8431 of *LNCS*, pages 97–118. Springer, 2014.

[43] M. A. G. de Dios, C. Dania, and M. Clavel. Formal reasoning about fine-grained access control policies. In H. Köhler and M. Saeki, editors, *Proceedings of the 11th Asia-Pacific Conference on Conceptual Modelling (APCCM 2015), Sydney, Australia, January 2015*, volume 165 of *Conference in Research and Practice in Information Technology (CRPIT)*. Australian Computer Society, 2015.

[44] M. A. G. de Dios, C. Dania, M. Schläpfer, D. Basin, M. Clavel, and M. Egea. SSG: a model-based development environment for smart, security-aware GUIs. In J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, pages 311–312. ACM, 2010.

[45] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Proceedings*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[46] B. Demuth. The OCL Portal. `http://www-st.inf.tu-dresden.de/ocl/`, 2005.

[47] B. Demuth, H. Hussmann, and S. Loecher. OCL as a specification language for business rules in database applications. In M. Gogolla and C. Kobryn, editors, *UML 2001: 4th International Conference on the Unified Modeling Language*, volume 2185 of *LNCS*, Toronto, Canada, 2001. Springer.

[48] B. Dutertre and L. Moura. Yices: An SMT solver. `http://yices.csl.sri.com/`, 2008.

[49] M. Egea. *An executable formal semantics for OCL with Applications to Formal Analysis and Validation*. PhD thesis, Universidad Complutense de Madrid, 2008.

[50] M. Egea, C. Dania, and M. Clavel. MySQL4OCL: A stored procedure-based MySQL code generator for OCL. *Electronic Communications of the EASST*, 36, 2010.

[51] M. Elkoutbi, I. Khriss, and R. Keller. Automated prototyping of user interfaces based on UML scenarios. *Automated Software Engineering*, 13:5–40, 2006.

[52] D. F. Ferraiolo, R. S. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

[53] E. Gagnon. Sablecc, 2014. `http://sablecc.org/`.

[54] M. Gogolla, M. Kuhlmann, and F. Büttner. A benchmark for ocl engine accuracy, determinateness, and efficiency. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, volume 5301 of *LNCS*, pages 446–459. Springer, 2008.

[55] M. Gogolla, M. Kuhlmann, and F. Büttner. Sources for a benchmark for OCL engine accuracy, determinateness, and efficiency, 2008.

[56] D. Gornik. A UML data modeling profile. `http://www.jeckle.de/files/RationalUML-RDB-Profile.pdf`, February 2005. IBM.

[57] F. Heidenreich. OCL-Codegenerierung für Deklarative Sprachen. Master's thesis, University of Dresden, March 2006. `http://www.dresden-ocl.org/index.php/DresdenOCL:Publications`.

[58] F. Heidenreich, C. Wende, and B. Demuth. A framework for generating query language code from OCL invariants. In *7th OCL Workshop at the UML/MoDELS Conference*, 2007.

[59] K. Hussey. MDT-OCL. `http://www.eclipse.org/modeling/mdt/?project=ocl`, 2008.

[60] X. Jia, A. Steele, L. Qin, H. Liu, and C. Jones. Executable visual software modeling—the ZOOM approach. *Software Quality Control*, 15:27–51, March 2007.

[61] A. Kleppe, W. Bast, J. B. Warmer, and A. Watson. *MDA Explained: The Model Driven Architecture–Practice and Promise*. Addison-Wesley, 2003.

[62] A. Konermann. The parser subsystem of the Dresden OCL 2.0 Toolkit - design and implementation. `http://www.dresden-ocl.org/index.php/DresdenOCL:Publications`, 2005.

[63] C. Kroiss, N. Koch, and A. Knapp. UWE4JSF: A model-driven generation approach for web applications. In M. Gaedke, M. Grossniklaus, and O. Díaz, editors, *Proc. of ICWE'09*, volume 5648 of *LNCS*, pages 493–496. Springer, 2009.

[64] V. Kulkarni, S. Reddy, and A. Rajbhoj. Scaling up model driven engineering - experience and lessons learnt. In D. C. Petriu, N. Rouquette, and Ø. Haugen, editors, *Part II of Proc. of Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010.*, LNCS, pages 331–345. Springer, 2010.

[65] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. López-Jaquero. UsiXML: A language supporting multi-path development of user interfaces. In R. Bastide, P. A. Palanque, and J. Roth, editors, *Engineering Human Computer Interaction and Interactive Systems, Joint Working Conferences EHCI-DSVIS 2004, Hamburg, Germany, July 11-13, 2004, Revised Selected Papers*, volume 3425 of *LNCS*, pages 200–220. Springer, 2004.

[66] T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-based modeling language for model-driven security. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *Proceedings of the 5th international Conference on the Unified Modeling Language: Model Engineering, Concepts, and Tools (UML'02)*, volume 2460 of *LNCS*, pages 426–441. Springer-Verlag, 2002.

[67] W. McCune. Prover9. `http://www.cs.unm.edu/~mccune/prover9/manual/June-2006C/`, 2006.

[68] Microsoft. Visual studio lightswitch, 2010. `http://www.microsoft.com/visualstudio/en-us/lightswitch`.

[69] P. J. Molina, S. Meliá, and O. Pastor. Just-UI : A user interface specification model. In C. Kolski and J. Vanderdonckt, editors, *Proc. of CADUI'02*, pages 63–74, 2002.

[70] Mozilla Foundation. XML user interface language (XUL), 2013. `https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XUL`.

[71] NESSoS. European Network of Excellence on Engineering Secure Future Internet Software Services and Systems, 2014. `http://www.nessos-project.eu/`.

[72] Object Management Group. Object constraint language specification version 2.0. Technical report, OMG, 2006. `http://www.omg.org/spec/OCL/2.0/`.

[73] Object Management Group. Object constraint language specification version 2.4. Technical report, OMG, 2014. `http://www.omg.org/spec/OCL/2.4/`.

[74] O. Pastor, J. Gómez, E. Insfrán, and V. Pelechano. The OO-method approach for information systems modeling: from object-oriented conceptual modeling to automated programming. *Information Systems*, 26(7):507–534, 2001.

[75] S. Sambasivam and P. Crefcoeur. How databases and XML can be used to master UML models, an investigation. *J. Comput. Small Coll.*, 23(6):220–228, 2008.

[76] M. Schläpfer. Automatic generation of smart SecGUIs from security design models. Master's thesis, Department of Computer Science, Information Security Division, ETH., Zurich, Switzerland, 2009.

[77] M. Schläpfer, M. Egea, D. Basin, and M. Clavel. Automatic generation of security-aware GUI models. In A. Bagnato, editor, *European Workshop on Security in Model Driven Architecture 2009 (SECMDA 2009), Enschede (The Netherlands), June 24, 2009. Proceedings.*, CTIT proceedings WP09-06, pages 42–56. Centre for Telematics and Information Technology, University of Twente, June 2009.

[78] A. Schramm, A. Preußner, M. Heinrich, and L. Vogel. Rapid UI development for enterprise applications: Combining manual and model-driven techniques. In D. C. Petriu, N. Rouquette, and Ø. Haugen, editors, *Part I of Proc. of Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010.*, LNCS, pages 271–285. Springer, 2010.

[79] SecProver, 2014. `http://actiongui.org/`, see SecProver project.

[80] Y. Shuxin and R. Indrakshi. Relational database operations modeling with UML. In *AINA '05: Proceedings of the 19th International Conference on Advanced Information Networking and Applications*, pages 927–932, Washington, DC, USA, 2005. IEEE Computer Society.

[81] Software Technology Group. The OCL 2.0 Dresden toolkit. `http://www.dresden-ocl.org/index.php/DresdenOCL:DresdenOCL2Toolkit`, 2007.

[82] Vaadin Ltd. Vaadin – a web application framework for rich internet applications, 2014. `http://vaadin.com/home`.

[83] J. Valdazo. Developing secure business applications from secure BPMN models. Master's thesis, Facultad de Informática, Universidad Complutense de Madrid, Madrid, Spain, 2012.

[84] Web Models Company. Web Ratio – you think, you get, 2015. `http://www.webratio.com`.

[85] WebRatio. Interaction flow modeling language (ifml) version 1.0 beta 2. Technical report, OMG, March 2014. `http://www.omg.org/spec/IFML/1.0/Beta2/`.

[86] J. Woodcock and J. Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

Part III

Papers related to this thesis

List of publications

- M. A. García de Dios, C. Dania, M. Clavel. "Formal reasoning about fine-grained access control policies". In *Proceedings of the 11th Asia-Pacific Conference on Conceptual Modelling (APCCM 2015), Sydney, Australia, January 2015*, Conference in Research and Practice in Information Technology (CRPIT), volume 165. Australian Computer Society. 2015.[43]

  APCCM is rated as a CORE-B conference by the Computing Research and Education Association of Australasia - CORE.

  The author main contributions are the following: study of the state of the art; design and implementation of the reasoning tool; collaboration in the reported case study.

- D. Basin, M. Clavel, M. Egea, M. A. García de Dios, C. Dania. "A Model-Driven Methodology for Developing Secure Data-Management Applications". *IEEE Transactions on Software Engineering*, 40(4):324–337, 2014.[12]

  Impact index: 2.59 en 2012 (1.98 en 2011, 2.22 en 2010). TSE is ranked 1st among Software Engineering journals.

  The author main contributions are the following: extensions of the security modeling language; definition of the GUI modeling language; the need for a transaction semantics for events; first implementation of the model editors and code generator; reported case studies.

- M. A. García de Dios, C. Dania, M. Clavel, D. Basin. "Model-driven development of a secure eHealth application." In *Engineering Secure Future Internet Services*. volume 8431 of *Lecture Notes in Computer Science*, pages 97–118. Springer, 2014.[42]

  This volume is a collection of selected results from the NESSoS Project. The submissions were reviewed by project members.

  The author main contribution is the full development of the reported case study: a secure eHealth application.

- D. Basin, M. Clavel, M. Egea, M. A. García de Dios, C. Dania. "ActionGUI semantics." IMDEA & ETH. Technical report. 2013.[6]

  This report contains the technical definitions that were not included in [12].

  The author main contributions are the following: extensions of the security modeling language; definition of the GUI modeling language; the need for a transaction semantics for events.

- D. Basin, M. Clavel, M. Egea, M. A. García de Dios, C. Dania, G. Ortiz, J. Valdazo. "Model-Driven Development of Security-Aware GUIs for Data-Centric Applications". In *Foundations of Security Analysis and Design VI - FOSAD Tutorial Lec-*

tures, volume 6858 of *Lecture Notes in Computer Science*, pages 101-124. Springer, 2011.[10]

This volume is a selection of the invited tutorial lectures given in the FOSAD Summer School.

The author main contributions are the following: extensions of the security modeling language; definition of the GUI modeling language; reported case studies; tutorial and manual (used in the lectures).

- M. A. García de Dios, C. Dania, M. Schlapfer, D. Basin, M. Clavel, M. Egea. "SSG: a model-based development environment for smart, security-aware GUIs". In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE 2010. ACM, pp. 311-312. 2010.[44]

  This short article corresponds to the poster and demo given at the Informal Demo track at ICSE 2010. ICSE is considered an A+ conference in CORE Computer Science Rankings.

  The author main contribution is the architecture design and implementation of the reported development environment.

- M. Clavel, M. Egea, M. A. Garca de Dios. "Checking Unsatisfiability for OCL Constraints". In *Proceedings of 9th OCL 2009 Workshop at the UML/MoDELS Conferences. The Pragmatics of OCL And Other Textual Specification Languages.* ECEASST, volume 24. 2009.[34]

  This workshop is the main international event for the OCL community of researchers and practitioners. Each paper is reviewed by three PC members. The average number of submissions is 15, out of which 9 or 10 papers are accepted.

  The author main contribution is the implementation of the unsatisfiability checker and the case study.

- M. Clavel, M. Egea, M. A. Garca de Dios. "Building an Efficient Component for OCL Evaluation." *Proceedings 8th OCL Workshop at the UML/MoDELS Conferences. OCL Concepts and Tools: From Implementation to Evaluation and Comparison.* ECEASST, volume 15. 2008.[33]

  This workshop is the main international event for the OCL community of researchers and practitioners. Each paper is reviewed by three PC members. The average number of submissions is 15, out of which 9 or 10 papers are accepted.

  The author main contribution is the design and implementation of an efficient OCL evaluator, as well as the reported case studies.

Formal Reasoning about Fine-Grained Access Control Policies

Miguel A. García de Dios **Carolina Dania** **Manuel Clavel**

IMDEA Software Institute, Spain
{miguelangel.garcia,carolina.dania,manuel.clavel}@imdea.org

Abstract

Nowadays, most of the main database management systems offer, in one way or another, the possibility of protecting data using fine-grained access control (FGAC) policies, i.e., policies that depend on dynamic properties of the system state. Reasoning about FGAC policies typically amounts to answering questions about whether a security-related property holds in a (possibly infinite) set of system states. To carry out this reasoning, we propose a novel, tool-supported methodology, which consists in transforming the aforementioned questions about FGAC policies into satisfiability problems in first-order logic. In addition, we report on our experience using the Z3 Satisfiability Modulo Theory (SMT) solver for automatically checking the satisfiability of the first-order formulas generated by the tool implementing our methodology, called SecProver, for a non-trivial set of examples.

1 Introduction

In Role-Based Access Control (RBAC) (Ferraiolo et al. 2001), permissions specify which roles are allowed to perform given operations. These roles typically represent job functions within an organization. Users are granted permissions by being assigned to the appropriate roles based on their competencies and responsibilities in the organization. RBAC allows one to organize the roles in a hierarchy where roles can inherit permissions along the hierarchy. In this way, the security policy can be described in terms of the hierarchical structure of an organization. However, it is not possible in RBAC to specify *fine-grained access control* (FGAC) policies, i.e., policies that depend on dynamic properties of the system state, for example, to allow an operation only during weekdays.

SecureUML (Basin et al. 2006) is a modeling language for formalizing FGAC policies. It extends RBAC with *authorization constraints*, which allow one to specify constraints for granting permissions. Authorization constraints are formalized in SecureUML using the Object Constraint Language (OCL) (Object Management Group 2014). Using SecureUML, one can then model access control decisions that depend on two kinds of information:

1. *static information*, namely the assignments of users and permissions to roles, and the role hierarchy, and
2. *dynamic information*, namely the satisfaction of authorization constraints in the given system state.

SecureUML is currently supported by ActionGUI (Basin et al. 2014, ActionGUI 2012), a model-driven methodology for developing secure data-management applications. In ActionGUI, system developers proceed by modeling three different views of the desired application: its data model, security model, and GUI model. These models formalize respectively the application's data domain, authorization policy, and its graphical interface together with the application's behavior. Afterwards a model-transformation function lifts the policy specified by the security model to the GUI model. Finally, a code generator generates a multi-tier application, along with all support for fine-grained access control, from the security-aware GUI model.

In this paper we propose a novel methodology for carrying out formal reasoning about FGAC policies specified using SecureUML. Reasoning about FGAC policies typically amounts to answering questions about whether a security-related property holds in a (possibly infinite) set of states. The key component of our methodology is a mapping from OCL to first-order logic (Clavel et al. 2009, Dania & Clavel 2013), thanks to which we are able to transform the aforementioned questions about FGAC policies into satisfiability problems in first-order logic. Finally, to validate our methodology, we have implemented a tool, called SecProver (SecProver 2014), and we have applied the Z3 SMT solver (de Moura & Bjørner 2008) for automatically checking the satisfiability of the first-order formulas generated by SecProver, for a non-trivial set of security-related questions about SecureUML models.

Organization. In Section 2 we provide background information about SecureUML, and we also discuss its semantics and compare its expressiveness with that of other languages currently supported by commercial database management systems. In Section 3 we summarize the key principles underlying our mapping from OCL to first-order logic. Then, in Section 4, we explain how, using the aforementioned mapping, interesting questions about SecureUML models can be translated into satisfiability problems in first-order

This work is partially supported by the Spanish Ministry of Economy and Competitiveness Project "StrongSoft" (TIN2012-39391-C04-01 and TIN2012-39391-C04-04).

logic, and, in Section 5, we report on our experience using the Z3 SMT solver for automatically checking the satisfiability of the formulas generated by our methodology, for a non-trivial set of examples. We conclude the paper with sections on related work and future work.

2 Modeling Fine-Grained Access Control Policies

SecureUML (Basin et al. 2006) is a modeling language for specifying fine-grained access control policies (FGAC) for actions on protected resources. Using SecureUML one can model *roles* (with their hierarchies), *permissions*, *actions*, *resources*, and constraints on the permissions, which are called *authorization constraints*. SecureUML is, however, *generic* in that it leaves open the nature of the protected resources, i.e., whether these resources are data, business objects, processes, controller states, etc. Basin et al. (2006) initially combined SecureUML with a design modeling language based on class diagrams, called ComponentUML, and with a language based on state diagrams, called ControllerUML. More recently, Basin et al. (2014) have combined SecureUML with a language for modeling graphical user interfaces for data-centric applications, called ActionGUI. In this work, we will use the aforementioned combination of SecureUML with ComponentUML, called SecureUML+ComponentUML.

Next, we will explain, and illustrate with examples, the main concepts used when modeling with SecureUML+ComponentUML: namely, resources and actions; invariants; authorization constraints; and permissions. Also, we will briefly compare SecureUML+ComponentUML with other languages supported by commercial data management systems for specifying FGAC policies

2.1 Resources and Actions

ComponentUML provides a subset of UML class models where entities (classes) can be related by associations and may have attributes. In SecureUML+ComponentUML, the protected resources are the ComponentUML entities, along with their attributes and association-ends (but not the associations as such), and the actions that they offer to the actors are those shown in the following table:

Resource	Actions
Entity	create, delete
Attribute	read, update
Association-end	read, create, delete

Example 1 In Figure 1 we show a ComponentUML model, named `EmplBasic.dtm`. This model specifies that every employee may have a name, a surname, and a salary; that every employee may have a supervisor and may in turn supervise other employees; and that every employee may take one of two roles: `Worker` or `Supervisor`. In the terminology of ComponentUML, `Employee` is an *entity*; `name`, `surname`, `salary`, and `role` are *attributes*; `supervisedBy` and `supervises` are *association-ends*; and `Role` is an *enumerated class*. Notice that the association-end `supervises` has multiplicity `0..*`, meaning that an employee may supervise zero or more employees, while the association-end `supervisedBy` has multiplicity `0..1` meaning that an employee may have at most one supervisor.

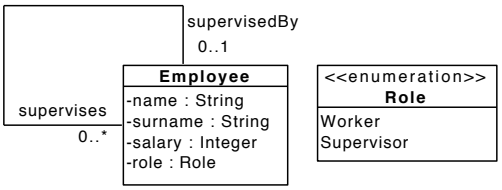
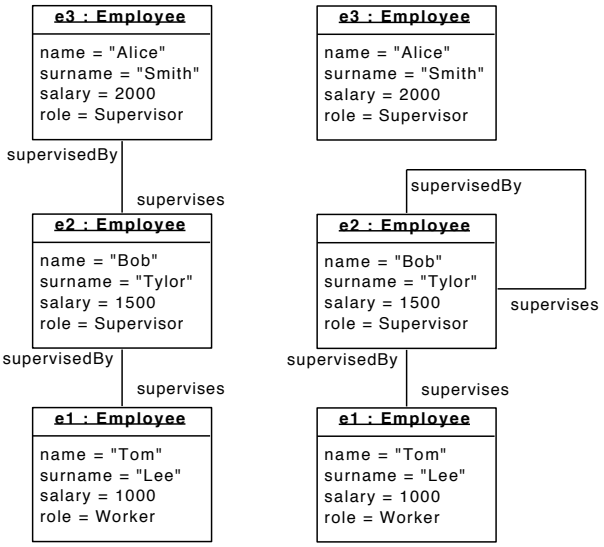


Figure 1: `EmplBasic.dtm`: a ComponentUML model for employees' information.



(a) Instance 1 (b) Instance 2

Figure 2: Two instances of `EmplBasic.dtm`

As expected, the *instances* of ComponentUML models are, basically, UML object models where objects can be related by links and can have values for their attributes.

Example 2 In Figure 2 we show two different instances of `EmplBasic.dtm`. In Instance 2a there are three employees, e_1, e_2 and e_3, and e_1 is supervised by e_2, e_2 is supervised by e_3, and e_3 has no supervisor at all. Moreover, e_1 has role `Worker` and both e_2 and e_3 have role `Supervisor`. Instance 2b has also three employees, e_1, e_2 and e_3, but this time e_1 is supervised by e_2, e_2 is supervised by itself, and e_3 has no supervisor at all. As before, e_1 has role `Worker` and both e_2 and e_3 have role `Supervisor`.

2.2 Invariants

ComponentUML models can be further refined by adding to them *invariants*, i.e., expressions specifying properties that every *valid* instance of the model must satisfy. Invariants are formalized in ComponentUML using the Object Constraint Language (OCL) (Object Management Group 2014).

OCL is a strongly typed, declarative language: expressions either have a primitive type (integer, real, string, or boolean), an entity type, a tuple type, or a collection type (set, bag, or collection). It provides standard operators on collections, such as →`isEmpty`, →`includes`, and →`excluding`, as well as operators to iterate over collections, such as →`forAll`, →`exists`, and →`select`. It also provides standard operators on primitive data and tuples, and a dot-operator to access the values of the objects' attributes and association-ends. Moreover, it includes

two constants, `null` and `invalid`, to represent *undefinedness*. Intuitively, `null` represents unknown or undefined values, whereas `invalid` represents error and exceptions. To check if a value is `null` or `invalid`, it provides, respectively, the boolean operators `oclIsUndefined()` and `oclIsInvalid()`.

Example 3 We can refine the model `EmplBasic.dtm` (Figure 1) by adding invariants to this model. In particular, consider the following constraints:

1. There is exactly one employee who has no supervisor.
2. Nobody is its own supervisor.
3. An employee has role `Supervisor` if and only if it has at least one supervisee.
4. Every employee has one role.

These constraints can be formalized in OCL as follows:

```
(1) Employee.allInstances()→one(e|
    e.supervisedBy.oclIsUndefined())
(2) Employee.allInstances()→forAll(e|
    not(e.supervisedBy = e))
(3) Employee.allInstances()→forAll(e|
    (e.role = Supervisor implies
        e.supervises->notEmpty())
    and (e.supervises->notEmpty()
        implies e.role = Supervisor))
(4) Employee.allInstances()→forAll(e|
    not(e.role.oclIsUndefined())
```

In what follows, we will refer to the constraint (1) as `oneBoss`, (2) as `noSelfSuper`, (3) as `roleSuper`, and (4) as `allRole`. Also, we will denote by `Empl1.dtm` the refined version of `EmplBasic.dtm` that includes as invariants the constraints `oneBoss`, `noSelfSuper`, `roleSuper`, and `allRole`. Notice that these four constraints evaluate to `true` in Instance 2a of `EmplBasic.dtm` (Figure 2), and therefore we say that this instance is a *valid* instance of `Empl1.dtm`. On the other hand, since `noSelfSuper` and `roleSuper` evaluate to `false` in Instance 2b of `EmplBasic.dtm` (Figure 2), we say that this other instance is a not a valid instance of `Empl1.dtm`.

2.3 Authorization Constraints

In SecureUML+ComponentUML, authorization constraints specify the conditions that need to be satisfied for a permission being granted to an actor (user) who requests it to perform an action. They are formalized using OCL, but they can also contain the following keywords:

- `self`: it refers to the root resource upon which the action will be performed, if the permission is granted. The root resource of an attribute or an association-end is the entity to which it belongs.
- `caller`: it refers to the actor that will perform the action, if the permission is granted.
- `value`: it refers to the value that will be used to update an attribute, if the permission is granted.
- `target`: it refers to the object that will be linked at the end of an association, if the permission is granted.

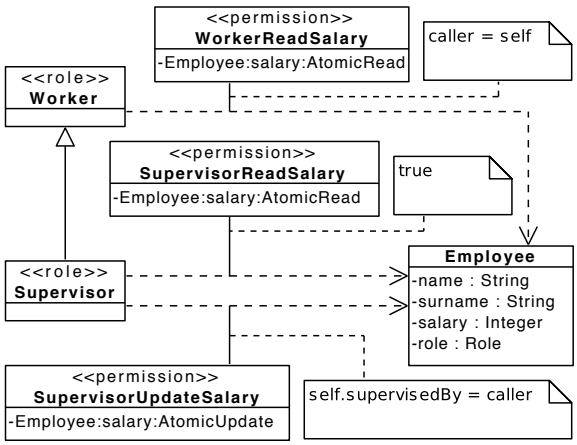


Figure 3: `Empl.stm`: a SecureUML+ComponentUML model for accessing employees' information.

Example 4 In Figure 3 we show a SecureUML+-ComponentUML model, named `Empl.stm`. This model specifies a basic FGAC policy for accessing the employees' information modeled in `Empl1.dtm`. Permissions are assigned to users depending on their *roles*, which can be `Worker` or `Supervisor`. Also, users with role `Supervisor` *inherit* all the permissions granted to users with role `Worker`, since `Supervisor` is a *subrole* of `Worker`. Finally, permissions are constrained by *authorization constraints*: namely,

1. A worker is granted permission to read an employee's salary, provided that it is its own salary, as specified by the authorization constraint `caller = self`.
2. A supervisor is granted unrestricted permission to read an employee's salary, as specified by the authorization constraint `true`.
3. A supervisor is granted permission to update an employee's salary, provided that it supervises this employee, as specified by the authorization constraint `self.supervisedBy = caller`.

2.4 Permissions

SecureUML+ComponentUML provides various syntactic sugar constructs for expressing FGAC policies in a more compact way. Basically, in the 'sweeter' presentation of a model, some roles may not have *explicitly* assigned any permission for some actions, while the following always holds in the de-sugared presentation of the model: every role has assigned exactly one permission for every action, and this permission has assigned exactly one authorization constraint.

Next, we will explain, and illustrate with examples, the rules that we apply for de-sugaring a SecureUML+ComponentUML model $\mathcal{S}$:

- *Role hierarchies.* Let *act* be an action and let r and r' be two roles. Suppose that r is a subrole of r' in $\mathcal{S}$, and that there is a permission in $\mathcal{S}$ for r' to execute *act* under the constraint *auth*. Then, when de-sugaring $\mathcal{S}$, we add a new permission to $\mathcal{S}$ for the role r to execute *act* under the same constraint *auth*.
- *Delete actions.* Let *entity* be an entity. Let *act* be the action delete(*entity*). Suppose that there is a permission in $\mathcal{S}$ for a role r to execute *act* under the constraint *auth*. Then, when

de-sugaring $\mathcal{S}$, for every association-end *assoc* owned by *entity*, we add to $\mathcal{S}$ a new permission for r to execute delete(*assoc*) under the same constraint *auth*.

- *Opposite association-ends.* Let *assoc* and *assoc*$'$ be two opposite association-ends. Let *act* be the action create(*assoc*). Suppose that there is a permission in $\mathcal{S}$ for a role r to execute *act* under the constraint *auth*. Then, when de-sugaring $\mathcal{S}$, we add to $\mathcal{S}$ a new permission for the role r to execute create(*assoc*$'$) under the constraint that results from replacing in *auth* the variable `self` by `target` and the variable `target` by `self`. De-sugaring is done similarly when *act* is the action delete(*assoc*).
- *Denying by default.* Let r be a role and let *act* be an action. Suppose that there is no permission in $\mathcal{S}$ for the role r to execute *act*. When de-sugaring $\mathcal{S}$, we add to $\mathcal{S}$ a new permission for the role r to execute *act* under the constraint `false`.
- *Disjunction of authorization constraints.* Let r be a role and let *act* be an action. Suppose that there are n permissions in $\mathcal{S}$ for the role r to execute *act*. When de-sugaring $\mathcal{S}$, we replace these n permissions by a new permission and assign to it the authorization constraint that results from disjoining together all the authorization constraints of the original n individual permissions.

In what follows, we will denote by Auth($\mathcal{S}, r$, *act*) the authorization constraint assigned, in the de-sugared presentation of the ComponentUML+SecureUML model $\mathcal{S}$, to the role r's permission for performing the action *act*.

Example 5 Consider the value of Auth($\mathcal{S}, r$, *act*) in the following cases:

Auth(`Empl.stm`, `Worker`, update(`salary`))= `false`,

by the rule "denying by default".

Auth(`Empl.stm`, `Supervisor`, update(`salary`))=
`self.supervisedBy = caller or false`,

by the combination of the rules "denying by default", "role hierarchies", and "disjunction of authorization constraints".

Auth(`Empl.stm`, `Worker`, read(`salary`))=
`caller = self`.

Auth(`Empl.stm`, `Supervisor`, read(`salary`))=
`caller = self or true`,

by the combination of the rules "denying by default", "role hierarchies", and "disjunction of authorization constraints".

2.5 Expressiveness

Traditionally, database management systems (DBMS) support 'all-or-nothing' access control with respect to the cells in the column of a table, i.e., a policy will either give or deny access to all the cells in the column. Nowadays, however, some of the main commercial DBMS also support fine-grained access control, which means that a policy can also give access only to a subset of the cells of the column. Next, we will provide some initial comparison between SecureUML+ComponentUML and the languages currently supported by Oracle Virtual Private Database (Huey 2014), IBM/DB2 (IBM 2013), Microsoft SQL Server (SQL 2012), and Teradata (Teradata 2014) for specifying FGAC policies.

Oracle supports FGAC in its Virtual Private Database (VPD) through the use of *security policy functions* (SPF), which are written in Oracle PL/SQL. The idea is that when a user executes a statement, the corresponding SPF is transformed into a `WHERE` clause and is added to the user's original statement. Clearly, authorization constraints play the same role as SPFs, and we conjecture, based on our experience mapping OCL into SQL (Egea et al. 2010), that any SPFs written in declarative SQL could be formalized as an authorization constraint written in OCL. However, since SPFs are written in PL/SQL, they would typically contain non-declarative code.

IBM/DB2 implements FGAC through the use of *row access control* and *column access control rules*. They specify, respectively, which rows and columns can be accessed and under which conditions. Again, authorization constraints play the same role as row and column access rules, and we also conjecture that any combination of row and column access control rules written in declarative SQL could be formalized as an authorization constraint written in OCL. On the other hand, and differently from SecureUML+ComponentUML, IBM/DB2 only supports column access control rules for `SELECT` statements, and, therefore, they can only be used, in general, to protect read-actions over attributes.

Finally, both Microsoft SQL Server and Teradata support FGAC policies through the use of *security labels*, which can be assigned to users and resources, and *constraints*. In SecureUML+ComponentUML, security labels can be represented as additional attributes of the entities representing users and resources, and constraints can then be formalized as OCL authorization constraints referring to the values of these additional attributes. On the other hand, security labels can only be assigned to entities, and therefore, they can not be used to protect read- or update-actions over attributes.

3 Mapping OCL to First-Order Logic

In previous work (Clavel et al. 2009, Dania & Clavel 2013) we proposed a mapping from OCL to first-order logic, which consists of two, inter-related components: (i) a map from ComponentUML models and boolean OCL expressions to first-order formulas, called ocl2fol$_{\text{def}}$; and (ii) a map from boolean OCL expressions to first-order formulas, called ocl2fol. The following remark formalizes the main property of our mapping from OCL to first-order logic.

Remark 1 Let $\mathcal{D}$ be a ComponentUML model, with invariants $expr_1, \ldots, expr_n$, and let *expr* be a boolean expression. Then, *expr* evaluates to `true` in every valid instance of $\mathcal{D}$ if and only if

$$\begin{aligned}&\text{ocl2fol}_{\text{def}}(\mathcal{D})\\ &\quad\cup \bigcup_{i=1}^{n} \text{ocl2fol}_{\text{def}}(expr_i) \cup \bigcup_{i=1}^{n} \{\text{ocl2fol}(expr_i)\}\\ &\quad\cup \text{ocl2fol}_{\text{def}}(expr) \cup \{\neg(\text{ocl2fol}(expr))\}\end{aligned}$$

is unsatisfiable.

Next, we will explain, and illustrate with examples, the main ideas behind the maps ocl2fol$_{\text{def}}$ and ocl2fol. We refer the interested reader to the original

papers (Clavel et al. 2009, Dania & Clavel 2013) for a more formal presentation of these maps and of the subset of OCL that they currently support.

3.1 The map ocl2fol$_{\text{def}}$ (models)

In our mapping from OCL to first-order logic, we represent entities by predicates, attributes by functions, and association-ends, depending on their multiplicity, either by binary predicates or by functions. Also, we represent `null` and `invalid`, respectively, by the constants null and invalid, and we introduce two unary predicates IsNull and IsInvalid, to represent when an element is `null` or `invalid`.

Let $\mathcal{D}$ be a ComponentUML model. ocl2fol$_{\text{def}}(\mathcal{D})$ returns the axioms formalizing the properties of the predicates and functions that represent the entities, attributes and association-ends in $\mathcal{D}$, as well as the axioms formalizing the constants null and invalid, and the predicates IsNull and IsInvalid.

Example 6 Consider the ComponentUML model `EmplBasic.dtm` shown in Figure 1. Among others, ocl2fol$_{\text{def}}$(`EmplBasic.dtm`) returns the axiom

$$\forall(x)(\text{Employee}(x) \Rightarrow \neg(\text{isNull}(x) \vee \text{isInvalid}(x))),$$

which formalizes that neither `null` nor `invalid` are objects of the entity `Employee`, as well as the axiom

$$\forall(x)\forall(y)(\text{supervises}(y,x) \Rightarrow (\text{supervisedBy}(x) = y)),$$

which formalizes the key property of `supervises` as the opposite association-end of `supervisedBy`.

The following remark formalizes the main property of the map ocl2fol$_{\text{def}}$.

Remark 2 Let $\mathcal{D}$ be a ComponentUML model. Then, there is a one-to-one correspondence between the instances of $\mathcal{D}$ and the first-order models that satisfy ocl2fol$_{\text{def}}(\mathcal{D})$.

3.2 The map ocl2fol

In our mapping from OCL to first-order logic, we represent boolean expressions as formulas.

Let *expr* be a boolean expression. ocl2fol(*expr*) is defined recursively over the structure *expr*, according to the following principles:

- Each subexpression C`.allInstances()` is represented by a predicate formula whose predicate is the one representing the entity C.
- Each boolean subexpression is represented by a formula which mirrors its logical structure.
- Each integer subexpression is represented by the corresponding functional expression.
- Each set subexpression is represented by a predicate formula whose predicate's definition is generated using the map ocl2fol$_{\text{def}}$ (see below).

Example 7 Consider the constraints `oneBoss` and `noSelfSuper` introduced in Example 3. ocl2fol(`oneBoss`) returns the formula

$$\begin{aligned}\exists(e)(&\text{Employee}(e) \wedge \text{isNull}(\text{supervisedBy}(e)) \\ &\wedge \forall(e')(\text{Employee}(e') \wedge \text{isNull}(\text{supervisedBy}(e')) \\ &\qquad \Rightarrow\ e' = e)),\end{aligned}$$

and ocl2fol(`noSelfSuper`) returns the formula

$$\forall(e)(\text{Employee}(e) \Rightarrow \neg(\text{supervisedBy}(e) = e)).$$

The following remark formalizes the main property of the map ocl2fol.

Remark 3 Let $\mathcal{D}$ be a ComponentUML model. Let *expr* be a boolean expression. Suppose that *expr* does not contain any subexpression of type collection. Then, there is a one-to-one correspondence between the instances of $\mathcal{D}$ in which the *expr* evaluates to `true` and the first-order models that satisfy ocl2fol$_{\text{def}}(\mathcal{D}) \cup \{\text{ocl2fol}(expr)\}$.

3.3 The map ocl2fol$_{\text{def}}$ (expressions)

Often, OCL expressions include subexpressions that will evaluate to collections: e.g., those which are built using →`select`, →`collect`, or →`excluding`. In our mapping from OCL to first-order logic, when these subexpressions are of type set, we represent them using new predicates whose definitions, which follow the principles underlying ocl2fol$_{\text{def}}$, are given by the map ocl2fol$_{\text{def}}$.

Example 8 Consider the expression

```
Employee.allInstances()→select(e|
    e.supervises→notEmpty()).
```

This expression, which we refer to as `colOfSuper`, will evaluate to a set containing only those employees whose list of supervisees is not empty. Then, ocl2fol$_{\text{def}}$(`colOfSuper`) returns the following axiom,

$$\begin{aligned}\forall(x)(&\text{P\_colOfSuper}(x) \Leftrightarrow \\ &(\text{Employee}(x) \wedge \exists(y)(\text{supervises}(x,y)))),\end{aligned}$$

where the new predicate P_colOfSuper represents the set that will be returned when evaluating `colOfSuper`.

The following remark formalizes the main property of the map ocl2fol$_{\text{def}}$ over expressions of type set.

Remark 4 Let $\mathcal{D}$ be a ComponentUML model. Let *expr* be an expression of type set. Let P_*expr* be the predicate symbol generated by ocl2fol(*expr*). Then, there is a one-to-one correspondence between the instances of $\mathcal{D}$ and the first-order models that satisfy ocl2fol$_{\text{def}}(\mathcal{D}) \cup$ ocl2fol$_{\text{def}}$(*expr*) for which the following holds: *expr* evaluates to $\{o_1, \ldots, o_n\}$ in $\mathcal{I}$ if and only if the element that corresponds to o_i belongs to the set that interprets P_*expr*, for $i = 1, \ldots, n$.

3.4 Reasoning about Data Models

Here we provide a simple example of the use of our mapping from OCL to first-order logic for reasoning about ComponentUML models.

In what follows, when a ComponentUML model $\mathcal{D}$ contains invariants $expr_1, \ldots, expr_n$, we will consider that ocl2fol$_{\text{def}}(\mathcal{D})$ includes also the formulas $\bigcup_{i=1}^{n}$ ocl2fol$_{\text{def}}$($expr_i$).

Example 9 Consider the following question about the model `Empl1.dtm` in Example 3: *Is there a valid instance in which someone is supervised by one of its own supervisees?* Let us formalize the property that no employee is supervised by their own supervisees as follows:

```
Employee.allInstances()→forAll(e|
    e.supervises→excludes(e.supervisedBy)).
```

We will refer to this expression as `noMixSuper`. Then, according to Remark 1, the answer to our question is 'Yes' since

$$\text{ocl2fol}_{\text{def}}(\texttt{Empl1.dtm}) \cup \{\neg(\text{ocl2fol}(\texttt{noMixSuper}))\}.$$

is satisfiable. Indeed, consider, for example, an instance of `Empl1.dtm` with just four employees, e_1, e_2, e_3, and e_4, such that e_1 is linked through the association-end `supervisedBy` with e_4, and similarly e_3 with e_2, and e_2 with e_3. Suppose also that e_1 is of role `Worker`, and e_2, e_3, and e_4 are of role `Supervisor`. This instance is certainly a valid one, since all the invariants evaluate to `true`. However, the expression `noMixSuper` evaluates to `false` because e_2 is linked through `supervisedBy` with e_3, but at the same time e_2 is also linked through the association-end `supervises` with e_3 (since e_3 is linked through `supervisedBy` with e_2).

4 Reasoning about Fine-Grained Access Control Policies

As discussed by Basin et al. (2014), SecureUML+ComponentUML models have a rigorous semantics. In particular, let $\mathcal{S}$ be a SecureUML+ComponentUML model and let $\mathcal{I}$ be an instance of its underlying data model. Also, let u be a user, with role r, and let *act* be an action, with arguments *args*. Then, according to the semantics of SecureUML+ComponentUML, $\mathcal{S}$ authorizes u to execute *act* in $\mathcal{I}$ if and only if $[\text{Auth}(\mathcal{S}, r, act)]^{(u,args)}$ evaluates to `true` in $\mathcal{I}$, where $[\text{Auth}(\mathcal{S}, r, act)]^{(u,args)}$ is the expression that results from replacing in $\text{Auth}(\mathcal{S}, r, act)$ the keyword `caller` by u, and the keywords `self`, `value`, and `target` by the corresponding values in *args*.

In what follows, given a SecureUML+ComponentUML model $\mathcal{S}$, we use the term *scenario* to refer to any valid instance of $\mathcal{S}$'s underlying data model in which a user requests permission to execute an action. For the sake of simplicity, we will assume that neither the user requesting permission nor the resource upon which the action will be executed can be *undefined*.

Next, we will explain, and illustrate with examples, how one can use the mapping from OCL to first-order logic discussed in Section 3 to reason about SecureUML+ComponentUML models. Unless stated otherwise, all our examples refer to the SecureUML+ComponentUML model `Empl.stm` (Example 4). Recall that this model's underlying data model is the ComponentUML model `Empl1.dtm` (Example 3), which includes the invariants `oneBoss`, `noSelfSuper`, `roleSuper`, and `allRole`.

We organize our examples in blocks or categories. In the first block, we are interested in knowing if there is any scenario in which someone with role r will be allowed to execute an action *act*. Notice that, by Remark 1, the answer will be 'No' if and only if the following set of formulas is unsatisfiable:

$$\text{ocl2fol}_{\text{def}}(\mathcal{D}) \cup \{\exists(caller)\exists(self)\exists(target)\exists(value)$$
$$(\text{ocl2fol}(caller.\texttt{role} = r) \wedge \text{ocl2fol}(\text{Auth}(\mathcal{S}, r, act)))\}.$$

Example 10 Consider the following question: *Is there any scenario in which someone with role* `Worker` *is allowed to change the salary of someone else (including itself)?* Recall that

$$\text{Auth}(\texttt{Empl.stm}, \texttt{Worker}, \text{update}(\texttt{salary})) = \texttt{false}.$$

According to Remark 1, the answer to this question is 'No', since the following set of formulas is clearly unsatisfiable:

$$\text{ocl2fol}_{\text{def}}(\texttt{Empl1.dtm}) \cup \{\exists(caller)\exists(self)$$
$$(\text{ocl2fol}(caller.\texttt{role} = \texttt{Worker})$$
$$\wedge \text{ocl2fol}(\texttt{false}))\},$$

(Note that ocl2fol(`false`) returns $\bot$.) Indeed, there is no scenario in which the expression `false` can evaluate to `true`.

Example 11 Consider the following question: *Is there any scenario in which someone with role* `Supervisor` *is allowed to change the salary of someone else (including itself)?* Recall that

$$\text{Auth}(\texttt{Empl.stm}, \texttt{Supervisor}, \text{update}(\texttt{salary})) =$$
$$(\texttt{self.supervisedBy = caller or false}).$$

According to Remark 1, the answer to this question is 'Yes', since the following set of formulas is satisfiable:

$$\text{ocl2fol}_{\text{def}}(\texttt{Empl1.dtm}) \cup \{\exists(caller)\exists(self)$$
$$(\text{ocl2fol}(caller.\texttt{role} = \texttt{Supervisor})$$
$$\wedge \text{ocl2fol}(self.\texttt{supervisedBy} = caller \texttt{ or false}))\}.$$

(Note that ocl2fol($self$.`supervisedBy` = $caller$) returns supervisedBy($self$) = $caller$). Consider, for example, a scenario with just two employees, e_1 and e_2, such that e_1 is linked with e_2 through the association-end `supervisedBy`. Suppose also that e_1 has role `Worker` and e_2 has role `Supervisor`. Clearly, for $caller = e_2$ and $self = e_1$, the expression $self$.`supervisedBy` = $caller$ evaluates to `true` in this scenario.

Example 12 Consider the following question: *Is there any scenario in which someone with role* `Supervisor` *is allowed to change its own salary?* Notice that in any scenario in which someone is requesting to change its own salary, the values of *self* (i.e., the employee whose salary is to be updated) and *caller* (i.e., the employee who is updating this salary) are the same. According to Remark 1, the answer to this question is 'No', since the following set of formulas is unsatisfiable:

$$\text{ocl2fol}_{\text{def}}(\texttt{Empl1.dtm}) \cup \{\exists(caller)\exists(self)$$
$$(\text{ocl2fol}(caller.\texttt{role} = \texttt{Supervisor})$$
$$\wedge \text{ocl2fol}(self = caller \texttt{ and}$$
$$(self.\texttt{supervisedBy} = caller \texttt{ or false})))\}.$$

Indeed, notice that, in every valid scenario the invariant `noSelfSuper` evaluates to `true`, which implies that there are no values for *caller* and *self* such that the expressions *self* = *caller* and *self*.`supervisedBy` = *caller* both evaluate to `true`.

Example 13 Consider the following question: *Is there any scenario in which someone with role* `Supervisor` *is allowed to change the salary of someone who has no supervisor at all?* Notice that in any scenario in which someone (*caller*) is requesting to change the salary of someone (*self*) who has no supervisor at all, the value of *self*.`supervisedBy` must be `null`. According to Remark 1, the answer to this question is 'No', since the following set of formulas is unsatisfiable:

$$\text{ocl2fol}_{\text{def}}(\texttt{Empl1.dtm}) \cup \{\exists(caller)\exists(self)$$
$$(\text{ocl2fol}(caller.\texttt{role} = \texttt{Supervisor})$$
$$\wedge \text{ocl2fol}(self.\texttt{supervisedBy} = \texttt{null and}$$
$$(self.\texttt{supervisedBy} = caller \texttt{ or false})))\}.$$

Indeed, notice that, by assumption, *caller* is always a defined object, i.e., it can not be `null`, and therefore, if the expression *self*.`supervisedBy = null` evaluates to `true`, then the expression *self*.`supervisedBy =` *caller* evaluates to `false`.

In our second block of examples, we are interested in knowing if there is any scenario in which someone with role r will not be allowed to execute an action *act*. Notice that, by Remark 1, the answer will be 'No' if and only if the following set of formulas is unsatisfiable:

$$\text{ocl2fol}_{\text{def}}(\mathcal{D}) \cup \{\exists(caller)\exists(self)\exists(target)\exists(value)\\ (\text{ocl2fol}(caller.\texttt{role = } r) \wedge \neg(\text{Auth}(\mathcal{S}, r, act)))\}.$$

Example 14 Consider the following question: *Is there any scenario in which someone with role* `Supervisor` *is not allowed to change the salary of someone else (including itself)?* According to Remark 1, the answer to this question is 'Yes', since the following set of formulas is satisfiable:

$$\text{ocl2fol}_{\text{def}}(\texttt{Empl1.dtm}) \cup \{\exists(caller)\exists(self)\\ (\text{ocl2fol}(caller.\texttt{role = Supervisor}) \wedge\\ \neg(\text{ocl2fol}(self.\texttt{supervisedBy = } caller \texttt{ or false})))\}.$$

Consider, for example, a scenario with just three employees, e_1, e_2, and e_3 such that e_1 is linked with e_2 through the association-end `supervisedBy`, and similarly e_2 with e_3; but e_1 is not linked with e_3 through the association-end `supervisedBy`. Suppose that e_2 and e_3 have role `Supervisor` and e_1 has role `Worker`. Clearly, for *caller* $= e_3$ and *self* $= e_1$, the expression *self*.`supervisedBy =` *caller* evaluates to `false` in this scenario.

In our third block of examples, we are interested in knowing if there is any scenario in which nobody with role r will be allowed to execute an action *act*. Notice that, by Remark 1, the answer will be 'No' if and only if the following set of formulas is unsatisfiable:

$$\text{ocl2fol}_{\text{def}}(\mathcal{D}) \cup \{\exists(self)\exists(target)\exists(value)\forall(caller)\\ (\text{ocl2fol}(caller.\texttt{role = } r) \Rightarrow\\ \neg(\text{ocl2fol}(\text{Auth}(\mathcal{S}, r, act))))\}.$$

Example 15 Consider the following question: *Is there any scenario in which nobody with role* `Supervisor` *is allowed to change the salary of someone else (including itself)*? According to Remark 1, the answer to this question is 'Yes', since the following set of formulas, which we will refer to as Forms(Ex 15), is satisfiable:

$$\text{ocl2fol}_{\text{def}}(\texttt{Empl1.dtm}) \cup \{\exists(self)\forall(caller)\\ (\text{ocl2fol}(caller.\texttt{role = Supervisor}) \Rightarrow\\ \neg(\text{ocl2fol}(self.\texttt{supervisedBy = } caller \texttt{ or false})))\}.$$

Indeed, consider, for example, a scenario with just two employees, e_1 and e_2, such that e_1 is linked with e_2 through the association-end `supervisedBy`. Suppose that e_1 has role `Worker` and e_2 has role `Supervisor`. Clearly, for *self* $= e_2$, for every value for *caller*, the expression *self*.`supervisedBy` $=$ *caller* evaluates to `false`.

In our fourth block of examples, we are interested in knowing if, in every scenario, there is at least one object upon which nobody with role r will be allowed to execute an action *act*. Notice that, by Remark 1, the answer will be 'Yes' if and only if the following set of formulas is unsatisfiable:

$$\text{ocl2fol}_{\text{def}}(\mathcal{D}) \cup \{\forall(self)\exists(target)\exists(value)\exists(caller)\\ (\text{ocl2fol}(caller.\texttt{role = } r) \wedge \text{ocl2fol}(\text{Auth}(\mathcal{S}, r, act)))\}.$$

Example 16 Consider the following question: *In every scenario, is there at least one employee whose salary can not be changed by anybody with role* `Supervisor`*?* According to Remark 1, the answer to this question is 'Yes', since the following set of formulas is unsatisfiable:

$$\text{ocl2fol}_{\text{def}}(\texttt{Empl1.dtm}) \cup \{\forall(self)\exists(caller)\\ (\text{ocl2fol}(caller.\texttt{role = Supervisor}) \wedge\\ \text{ocl2fol}(self.\texttt{supervisedBy = } caller \texttt{ or false}))\}.$$

Indeed, notice that in every valid scenario the invariant `oneBoss` evaluates to `true`, which means that there is one employee in the scenario who has no supervisor. In other words, for every valid scenario, we can find a value for *self* such that no value for *caller* can be found such that the expression *self*.`supervisedBy =` *caller* evaluates to `true`.

To end this section, we want to illustrate the importance of taking into account the invariants of the underlying data model when reasoning about FGAC policies. Let `Empl2.dtm` be the ComponentUML model that results from adding to the model `EmplBasic.dtm` (Example 1) the invariants `noSelfSuper`, `roleSuper`, `allRole`, plus the following invariant:

5. Everybody has one supervisor.

This invariant, which we will refer to as `allSuper`, can be formalized in OCL as follows:

```
Employee.allInstances()→forAll(e|
    not(e.supervisedBy.oclIsUndefined())).
```

Example 17 Consider the security model `Empl.stm` (Example 4), but this time with `Empl2.dtm` as its underlying data model. Consider again the question that we asked ourselves in Example 15: namely, *is there any scenario in which nobody with role* `Supervisor` *is allowed to change the salary of someone else (including itself)*? According to Remark 1, the answer to this question is different from Example 15, namely, 'No', since the set of formulas Forms(Ex 15) is now unsatisfiable. Indeed, notice that in every valid scenario the invariants `allSuper` and `roleSuper` both evaluate to `true`, which means that, for each value for *self*, we can find a value for *caller* such that the expressions *self*.`supervisedBy =` *caller* and *caller*.`role = Supervisor` both evaluate to `true`.

Finally, let `Empl3.dtm` be the ComponentUML model that results from removing from `Empl2.dtm`, the invariant `roleSuper`.

Example 18 Consider the security model `Empl.stm` (Example 4), but this time with `Empl3.dtm` as its underlying data model. Consider, once again, the question that we asked ourselves in Example 15: namely, *is there any scenario in which nobody with role* `Supervisor` *is allowed to change the salary of someone else (including itself)*? According to Remark 1, the answer to this question is now different

from Example 17, namely, 'Yes', since the set of formulas Forms(Ex 15) is now satisfiable. Indeed, consider a scenario with three employees e_1, e_2, and e_3, such that e_1 is linked with e_2 through the association-end `supervisedBy`, and similarly e_2 with e_3 and e_3 with e_1. Suppose also that e_2 and e_3 have role `Supervisor`, but e_1 has role `Worker`. (Notice that, since `roleSuper` is not included in `Empl3.dtm`, nothing prevents e_1 from not having the role `Supervisor`, despite the fact that it is linked with e_3 through the association-end `supervises`.) Clearly, for *self* $= e_3$, for every *caller* of role `Supervisor`, namely, e_2 and e_3, the expression *self*.`supervisedBy =` *caller* evaluates to `false`.

5 Automatically Reasoning about Fine-Grained Access Control Policies

Satisfiability modulo theories (SMT) solvers are tools for automatically proving the satisfiability of first-order formulas in a number of logical theories and their combination (Barrett et al. 2009). Basically, SMT generalizes boolean satisfiability (SAT) by incorporating equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other useful first-order theories. Of course, when dealing with quantifiers, SMT solvers cannot be complete, and may return *unknown* after a while, meaning that they can neither prove the quantified formula to be unsatisfiable, nor can they find an interpretation that makes it satisfiable. In the past years, there has been a great deal of interest and research on the foundational and practical aspects of SMT. They have also become the backbone of numerous applications in automated verification, artificial intelligence, program synthesis, security, product configuration, and much more.

We briefly report here on our experience using the Z3 SMT solver (de Moura & Bjørner 2008) to automatically obtain the answers to the questions posed in the examples in Section 4. Table 1 below summarizes the results of our experiments. For each example, we show the time it takes Z3 to return an answer (in all cases, less than 1 second); the answer that it returns (in all cases, the expected one); and the first-order model that it generates when the answer is `sat`, i.e., when it finds that the input set of formulas is satisfiable. Each model represents a scenario (not necessarily the one discussed in Section 4 for the corresponding example), and here we simply indicate the number of employees that it contains, which employees are linked through the association-end `supervisedBy`, which employees have the role `Worker`, which employees have the role `Supervisor`, which employee is the one requesting permission to change the salary (*caller*), and which employee is the one whose salary will be changed (*self*) if permission is granted. We ran our experiments on a laptop computer, with a 2.66GHz Intel Core 2 Duo processor and 4GB 1067 MHz memory, using Z3 version 4.3.2 9d221c037a95-x64-osx-10.9.2. Finally, the input for Z3 has been generated using our tool SecProver (SecProver 2014). This tool takes the following parameters: a data model, a security model, a set (possibly empty) of invariants, an action, a role, a set (possibly empty) of additional constraints, and a *question type*.[1] SecProver automatically generates the set of first-order formulas whose satisfiability will determine, according to our methodology, the answer to the given question.

[1]Currently, only four question types are supported, which correspond to the four blocks of examples considered in Section 4, but other question types will be added soon. The reason for using question types is to make it easier for those users who may not be familiar with first-order logic to understand the precise meaning of their questions, as well as the responses eventually given by the SMT solver to these questions.

Ex.	Time	Ans.	Interpretation
10	0.078s	`unsat`	—

11	0.107s	`sat`	#employees = 3
			supervisedBy = $\{(e_3, e_2), (e_1, e_2)\}$
			Worker = $\{e_1, e_3\}$
			Supervisor = $\{e_2\}$
			caller $= e_2$, *self* $= e_1$

12	0.041s	`unsat`	—
13	0.042s	`unsat`	—

14	0.306s	`sat`	#employees = 6
			supervisedBy = $\{(e_1, e_2), (e_2, e_3), (e_4, e_2), (e_5, e_3), (e_6, e_3)\}$
			Worker = $\{e_1, e_4, e_5, e_6\}$
			Supervisor = $\{e_2, e_3\}$
			caller $= e_3$, *self* $= e_1$

15	0.078s	`sat`	#employees = 1
			supervisedBy = $\emptyset$
			Worker = $\{e_1\}$
			Supervisor = $\emptyset$
			self $= e_1$

16	0.485s	`unsat`	—
17	0.060s	`unsat`	—

18	0.506s	`sat`	#employees = 15
			supervisedBy = $\{(e_1, e_2), (e_2, e_4), (e_3, e_4), (e_4, e_6), (e_5, e_4)(e_6, e_{12}), (e_7, e_4), (e_8, e_{14}), (e_9, e_4), (e_{10}, e_4), (e_{11}, e_{15}), (e_{12}, e_{13}), (e_{13}, e_4), (e_{14}, e_4), (e_{15}, e_4)\}$
			Worker = all
			Supervisor = $\emptyset$
			self $= e_2$

Table 1: Automatic reasoning over the examples 10-18 introduced in Section 4.

6 Related Work

Many proposals exist for reasoning about RBAC policies, each one using a different logic or formalism, including the so-called "default" logic (Woo & Lam 1993), modal logic (Massacci 1997), higher-order logic (Appel & Felten 1999), C-Datalog (Bacon et al. 2002), first-order logic (Jajodia et al. 2001, Bertino et al. 2003), and description logic (Zhao et al. 2005). To the best of our knowledge none of these proposals has been properly extended to cope with FGAC policies. In recent years, however, there has been a growing interest in finding appropriate formalisms and frameworks for specifying and analysing FGAC policies. In a nutshell, our proposal differs from other approaches in that: (i) we use Se-

cureUML+ComponentUML (Basin et al. 2006) for modeling FGAC policies, and (ii) we use a mapping from OCL to first-order (Clavel et al. 2009, Dania & Clavel 2013) for reasoning about these policies. In our opinion, our approach has two main advantages: (i) the reasoning about FGAC policies can take into account the properties of the system states, since OCL is the language that we use both for specifying the invariants in the data model and the authorization constraints in the security model; and (ii) the reasoning about FGAC policies can be done automatically (although sometimes may fail to find a result), since the mapping that we use for translating OCL into first-order logic supports the effective application of SMT solvers over the generated formulas.

Halpern & Weissman (2008) have proposed an interesting framework for specifying and reasoning about FGAC policies, called Lithium. It is based on a decidable fragment of (multi-sorted) first-order logic. Differently from OCL, this logic does not consider undefined values, which, based on our experience, is something crucial when formalizing properties of the system states. Unfortunately, we are not aware of case studies that have been carried out using Lithium, and which we could use to compare it with our approach in terms of the expressiveness of the underlying formalisms and of the effectiveness of the associated reasoning tools.

Kuhlmann et al. (2011, 2013) propose a domain-specific language for specifying role-based policies which is based on UML and OCL. For the purpose of analyzing these policies, they propose to use SAT solvers, and, in particular the one implemented in Alloy (Jackson 2002). Differently from SMT solvers, Alloy requires the search space to be bounded, by explicitly indicating the number of objects in each entity, the number of links of each association and the possible values of each attribute. Also, integer expressions are not allowed, neither in the invariants nor in the policies under consideration. On the other hand, this approach enables one to include, within the policies, some time-constraints, which are not possible in our approach.

Finally, in the context of XACML (OASIS 2013), there exists a XACML profile for the specification of RBAC policies (OASIS 2010). However, no methods have been proposed for reasoning about policies written with this profile. Also, it is unclear whether this profile can be extended to cope with fine-grained access control policies. To address the first concern, Helil & Rahman (2010) propose an extension of the XACML profile for RBAC based on OWL. This approach supports the use of an OWL-DL reasoner for deciding about RBAC policies within XACML. More interestingly, Ramli et al. (2014) have recently proposed a new syntax and semantics for XACML, for the purpose of supporting formal reasoning about XACML policies. One of the challenges here is to formalize the different algorithms for enforcing policy rules which are available in XACML. Ramli et al. (2014) formalize the majority of these algorithms, and propose two new algorithms (one of which is very close to the semantics of SecureUML+ComponentUML.) Another challenge is to formalize the concepts of obligations and advices in XACML, but they are not covered by Ramli et al. (2014). Finally, with respect to methods for reasoning about XACML policies, Ramli et al. (2014) propose to explore the use of SMT solvers, but no experiments are reported yet.

7 Conclusions and Future Work

Model-driven engineering supports the development of complex software systems by generating software from models. Model-driven security (Basin et al. 2011) is a specialization of this paradigm, where system designs are modeled together with their security requirements and security infrastructures are directly generated from the models. Of course, the quality of the generated code depends on the quality of the source models. If the models do not properly specify the system's intended behavior, one should not expect the generated system to do so either. Experience shows that even when using powerful, high-level modeling languages, it is easy to make logical errors and omissions. It is critical not only that the modeling language has a well-defined semantics, but also that there is tool support for analyzing the modeled systems' properties.

In this paper we have presented a novel, tool-supported methodology for reasoning about fine-grained access control policies (FGAC). We have also briefly reported on our experience using the Z3 SMT solver (de Moura & Bjørner 2008) for automatically proving non-trivial properties about FGAC policies. Within our methodology, we use SecureUML (Basin et al. 2006) to specify FGAC policies. SecureUML is a modeling language that extends role-based access control (RBAC) (Ferraiolo et al. 2001) with authorization constraints, which are formalized using the Object Constraint Language (OCL) (Object Management Group 2014).

The key component of our methodology is a mapping from OCL to first-order logic (Clavel et al. 2009, Dania & Clavel 2013), which allows one to transform questions about FGAC policies into satisfiability problems in first-order logic. Although this mapping does not cover the complete OCL language, our experience shows that the kind of OCL expressions typically used for specifying invariants and authorization constraints are covered by our mapping. More intriguing is, however, the issue about the effectiveness of SMT solvers for automatically reasoning about FGAC policies. Although our experience so far is extremely encouraging (all problems are solved in less than a second), we should not forget that our results completely depend on the interaction between (i) the way our mapping translates into first-order logic the relevant OCL expressions (invariants and authorization constraints) and (ii) the heuristics implemented in the SMT solver. We are currently analyzing this interaction in depth to better understand its scope and limitations. Ultimately, we know that there is a trade-off when using SMT solvers. On the one hand, they are necessarily incomplete and their results depend on heuristics, which may change. In fact, we have experienced (more than once) that two different versions of Z3 may return 'sat' and 'unknown' for the very same problem. This is not surprising (since two versions of the same SMT solver may implement two different heuristics) but it is certainly disconcerting. On the other hand, SMT solvers are capable of checking, in a fully automatic and very efficient way, the satisfiability of large sets of complex formulas. In fact, we have examples, involving more than a hundred non-trivial OCL expressions, which are checked by Z3 in just a few seconds.

Finally, as part of our future work, we plan to define formal mappings between the FGAC languages and frameworks supported by commercial DBMS (e.g., Oracle, IBM/DB2, Microsoft SQL Server and Teradata) and SecureUML. These mappings will allow us to apply our methodology also when reasoning

about FGAC policies in commercial DBMS.

References

ActionGUI (2012). http://actiongui.org/, see ActionGUI project.

Appel, A. W. & Felten, E. W. (1999), Proof-carrying authentication, *in* J. Motiwalla & G. Tsudik, eds, 'ACM Conference on Computer and Communications Security', ACM, pp. 52–62.

Bacon, J., Moody, K. & Yao, W. (2002), 'A model of OASIS role-based access control and its support for active security', *ACM Trans. Inf. Syst. Secur.* **5**(4), 492–540.

Barrett, C. W., Sebastiani, R., Seshia, S. A. & Tinelli, C. (2009), 'Satisfiability modulo theories.', *Handbook of satisfiability* **185**, 825–885.

Basin, D. A., Clavel, M. & Egea, M. (2011), A decade of model-driven security, *in* 'SACMAT 2011', Vol. 1998443, New York, NY, USA, Innsbruck, Austria, pp. 1–10.

Basin, D. A., Clavel, M., Egea, M., de Dios, M. A. G. & Dania, C. (2014), 'A model-driven methodology for developing secure data-management applications', *IEEE Trans. on Software Engineering* **40**(4), 324–337.

Basin, D., Doser, J. & Lodderstedt, T. (2006), 'Model driven security: from UML models to access control infrastructures.', *ACM Trans. on Software Engineering and Methodology* **15**(1), 39–91.

Bertino, E., Catania, B., Ferrari, E. & Perlasca, P. (2003), 'A logical framework for reasoning about access control models', *ACM Trans. Inf. Syst. Secur.* **6**(1), 71–127.

Clavel, M., Egea, M. & de Dios, M. A. G. (2009), 'Checking unsatisfiability for OCL constraints', *Electronic Communications of the EASST* **24**, 1–13.

Dania, C. & Clavel, M. (2013), OCL2FOL+: coping with undefinedness, *in* J. Cabot, M. Gogolla, I. Ráth & E. D. Willink, eds, 'OCL@MoDELS', Vol. 1092 of *CEUR Workshop Proceedings*, pp. 53–62.

de Moura, L. M. & Bjørner, N. (2008), Z3: an efficient SMT solver, *in* C. R. Ramakrishnan & J. Rehof, eds, 'Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Proceedings', Vol. 4963 of *LNCS*, Springer, pp. 337–340.

Egea, M., Dania, C. & Clavel, M. (2010), 'MySQL4OCL: a stored procedure-based MySQL code generator for OCL', *Electronic Communications of the EASST* **36**.

Ferraiolo, D. F., Sandhu, R. S., Gavrila, S., Kuhn, D. R. & Chandramouli, R. (2001), 'Proposed NIST standard for role-based access control', *ACM Trans. Inf. Syst. Sec.* **4**(3), 224–274.

Halpern, J. Y. & Weissman, V. (2008), 'Using first-order logic to reason about policies', *ACM Trans. Inf. Syst. Secur.* **11**(4).

Helil, N. & Rahman, K. (2010), 'Extending XACML profile for RBAC with semantic concepts'.

Huey, P. (2014), 'Oracle database security guide', http://docs.oracle.com/database/121/.

IBM (2013), 'IBM DB2. Database security guide', http://www-01.ibm.com/support/docview.wss?uid=swg27038855.

Jackson, D. (2002), Alloy: a new technology for software modelling, *in* J. Katoen & P. Stevens, eds, 'Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Proceedings', Vol. 2280 of *LNCS*, Springer, p. 20.

Jajodia, S., Samarati, P., Sapino, M. L. & Subrahmanian, V. S. (2001), 'Flexible support for multiple access control policies', *ACM Trans. Database Syst.* **26**(2), 214–260.

Kuhlmann, M., Sohr, K. & Gogolla, M. (2011), Comprehensive two-level analysis of static and dynamic RBAC constraints with UML and OCL, *in* 'Fifth International Conference on Secure Software Integration and Reliability Improvement, SSIRI 2011', IEEE, pp. 108–117.

Kuhlmann, M., Sohr, K. & Gogolla, M. (2013), 'Employing UML and OCL for designing and analysing role-based access control', *Mathematical Structures in Computer Science* **23**(4), 796–833.

Massacci, F. (1997), Reasoning about security: a logic and a decision method for role-based access control, *in* D. M. Gabbay, R. Kruse, A. Nonnengart & H. J. Ohlbach, eds, 'Qualitative and Quantitative Practical Reasoning, First International Joint Conference on Qualitative and Quantitative Practical Reasoning ECSQARU-FAPR'97, Proceedings', Vol. 1244 of *LNCS*, Springer, pp. 421–435.

OASIS (2010), 'XACML core and hierarchical role-based access control', http://docs.oasis-open.org/xacml/3.0/.

OASIS (2013), 'Extensible access control markup language (XACML)', http://docs.oasis-open.org/xacml/3.0/.

Object Management Group (2014), Object Constraint Language specification, Technical report, OMG. http://www.omg.org/spec/OCL/2.4.

Ramli, C. D. P. K., Nielson, H. R. & Nielson, F. (2014), 'The logic of XACML', *Sci. Comput. Program.* **83**, 80–105.

SecProver (2014). http://actiongui.org/, see SecProver project.

SQL (2012), 'Microsoft SQL Server 2012. Implementing row- and cell-level security in classified databases', http://msdn.microsoft.com/en-us/library/bb545450.aspx.

Teradata (2014), 'Teradata database. Security administration', http://www.info.teradata.com/.

Woo, T. Y. C. & Lam, S. S. (1993), 'Authorizations in distributed systems: A new approach', *Journal of Computer Security* **2**(2-3), 107–136.

Zhao, C., Heilili, N., Liu, S. & Lin, Z. (2005), Representation and reasoning on RBAC: a description logic approach, *in* D. V. Hung & M. Wirsing, eds, 'Theoretical Aspects of Computing - ICTAC 2005, Second International Colloquium, Proceedings', Vol. 3722 of *LNCS*, Springer, pp. 381–393.

A Model-Driven Methodology for Developing Secure Data-Management Applications

David Basin, Manuel Clavel, Marina Egea, Miguel A. García de Dios, and Carolina Dania

Abstract—We present a novel model-driven methodology for developing secure data-management applications. System developers proceed by modeling three different views of the desired application: its data model, security model, and GUI model. These models formalize respectively the application's data domain, authorization policy, and its graphical interface together with the application's behavior. Afterwards a model-transformation function lifts the policy specified by the security model to the GUI model. This allows a separation of concerns where behavior and security are specified separately, and subsequently combined to generate a security-aware GUI model. Finally, a code generator generates a multi-tier application, along with all support for access control, from the security-aware GUI model. We report on applications built using our approach and the associated tool.

Index Terms—Model-driven development, model-driven security, access control, GUI models, model transformation

1 Introduction

Data-management applications are focused around so-called CRUD actions that create, read, update, and delete data from persistent storage. These operations are the building blocks for numerous applications, for example dynamic websites where users create accounts, store and update information, and receive customized views based on their stored data. When the data managed is sensitive, then security is a concern and the use of these actions must be controlled.

Access control is the standard approach to restricting users' actions on data. When the access-control policies are sufficiently simple, it may be possible to formalize them declaratively, independent of the application's business logic. For example, multi-tier systems for web-based applications often build support for role-based access control (RBAC) into the application server, which is configured independently of the application's procedural details. In contrast, fine-grained access control policies may depend not only on the user's credentials but also on the satisfaction of constraints on the state of the persistence layer, i.e., on the values of stored data items. In such cases, authorization checks are typically implemented programmatically, by directly encoding them at appropriate places in the application. Unfortunately, these programmatic additions are cumbersome, error prone, and scale poorly. Moreover, they are difficult to audit and maintain as the authorization checks are spread throughout the code and security policy changes require code changes.

- *D. Basin is with ETH Zürich, Zürich, Switzerland. E-mail: basin@inf.ethz.ch.*
- *M. Clavel, M.A. García de Dios, and C. Dania are with IMDEA Software, Campus de Montegancedo, s/n, Pozuelo de Alarcon, 28223 Madrid, Spain. E-mail: {manuel.clavel, miguelangel.garcia, carolina.dania}@imdea.org.*
- *M. Egea is with ATOS Research & Innovation, Madrid, Spain. E-mail: marina.egea@atos.net.*

Manuscript received 30 May 2013; revised 17 Dec. 2013; accepted 24 Dec. 2013; date of publication 1 Jan. 2014; date of current version 1 May 2014. Recommended for acceptance by L. Williams.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2013.2297116

In this paper, we propose a methodology for the model-driven development of secure data-management applications. It consists of languages for modeling multi-tier systems, and a toolkit for generating these systems. Within our methodology, a secure data-management application is modeled using three interrelated models:

1. A *data model* defines the application's data domain in terms of its classes, attributes, associations, and (non-CRUD) methods;
2. A *security model* defines the application's security policy in terms of authorized access to the actions on the resources provided by the data model.
3. A graphical user interface, or *GUI model*, defines the application's graphical interface and application logic. Note, in particular, that this model formalizes both UI structure and behavior.

The heart of this methodology, illustrated in Fig. 1, is a model-transformation function that automatically lifts the policy that is specified in the security model to the GUI model. The idea is simple but powerful. The security model specifies under what conditions actions on data are authorized. The control information in the GUI model specifies which actions are executed in response to which events. Lifting essentially consists of prefixing each data action in the GUI model with the authorization check specified in the security model. The resulting GUI model is security aware. It specifies UI structure, information flow with persistent storage, and all authorization checks.

We have implemented this methodology within a toolkit, called ActionGUI [1], that performs this many-models-to-model transformation. From the resulting security-aware GUI model, ActionGUI generates a deployable application, along with all support for access control. In particular, when the security-aware GUI model contains only calls to execute CRUD actions, then ActionGUI will generate the complete implementation automatically.

The methodology and tool that we report on constitute a substantial further development of [2], [3]. In this previous

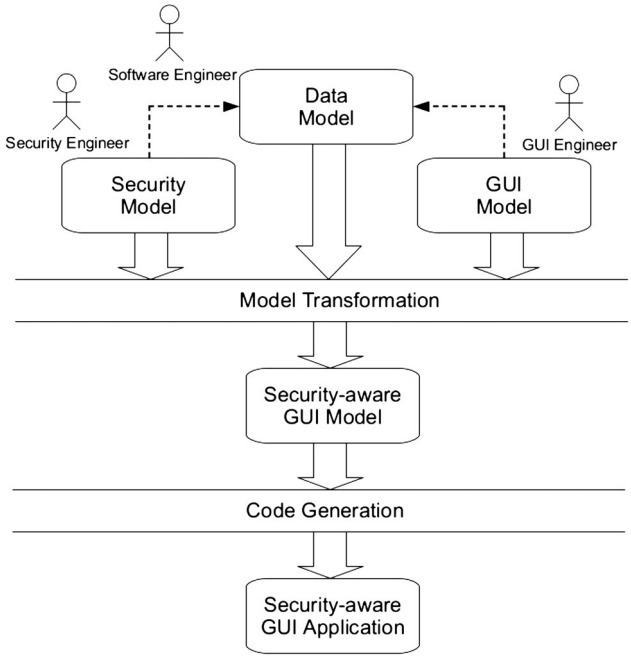


Fig. 1. Model-driven development of security-aware GUIs.

work, we proposed the idea of using model transformations to lift the security policy, formulated in terms of the data model to the GUI model. Here we improve and generalize this previous work and we provide an updated presentation of the modeling languages, the toolkit, and the example applications that we developed.

Let us expand on the main generalization with respect to our previous work. Lifting previously consisted of prefixing each event in the GUI model with the authorization check specified in the security model. However since the data actions executed by an event may change the persistence layer's state, checking authorization at the level of events, and therefore before executing any data action, is sufficient only if the underlying security policy does not contain authorization constraints (which was explicitly assumed in [3]), or if they do not depend on values that are changed during the execution of the event's data actions (as was the case in the examples discussed in [2]). To overcome this limitation, we now check authorization before executing each event's data actions and we provide events with a *transaction* semantics: either all of the data actions are executed in the given order, or none of them are executed at all.

Overall, we see our contributions as follows. First, our methodology offers Model Driven Architecture's purported benefits [4], [5] for data-management applications. By working with models, designers can focus on the application's data, behavior, security, and presentation, independent of the different, often complex, technologies that are used to implement them. Second, our use of model transformations leads to modularity and separation of concerns: the GUI model and the security model can be changed independently and by different developers, if desired. This avoids the problems mentioned earlier with fine-grained, hard-coded security policies that are difficult to maintain and audit. Finally, our methodology is quite powerful and compares favorably to alternatives, which are described in detail in Section 6 on related work. In particular, it leverages well-known security languages [6] for modeling rich, fine-grained access control policies, which must often be manually encoded in other proposals. Moreover, our new language for GUIs supports modeling realistic, dynamic web interfaces (where the web content varies based on the user's actions or user-provided data), without limiting the interfaces to a fixed set of templates or interaction patterns, as in other methodologies. Of course, the proof of the pudding is in the eating and we report on applications that we developed, which provide evidence of the applicability of this approach.

Organization. The remainder of this paper is organized as follows. In Section 2 we present background on the existing modeling languages that we use, namely, ComponentUML and SecureUML; in Appendix A we provide additional explanation on the semantics of the latter. In Section 3 we introduce a new modeling language, called GUIML, for modeling graphical user interfaces together with their behavior. In Section 4 we discuss our many-models-to-model transformation and in Section 5 we report on our tool support and on example applications that we developed using it. Finally, in Section 6 we survey related work and we draw conclusions in Section 7. Due to space limitations, we omit the formal account of our methodology, which is given in the technical report [7]. Instead, we provide in Appendix B a high-level account of the correctness of the model-transformation function, which lies at our methodology's core.

2 Background

For modeling an application's data and security policy, we leverage existing modeling languages, namely, ComponentUML and SecureUML [6]. In this section we briefly introduce these languages. Since SecureUML uses the Object Constraint Language (OCL) [8] to model authorization policies, we also summarize its main features.

2.1 ComponentUML

Data models provide a data-oriented view of a system. Typically they are used to specify how data is structured, the format of data items, and their logical organization, i.e., how data items are grouped and related. Our methodology employs ComponentUML for data modeling. ComponentUML provides a subset of UML class models where *entities* (classes) can be related by *associations* and may have *attributes* and *methods*. In ComponentUML, associations are binary: they always have two *association-ends* connecting two, not necessarily distinct, entities.

While ComponentUML and SecureUML have a graphical concrete syntax (see [6]), to simplify and clarify the presentation, we shall use textual concrete syntax. In this syntax, entities are declared with the keyword **Entity** followed by the entity's name, and its attributes and association-ends, which are enclosed within brackets. Attributes and association-ends are declared together with their types. Moreover, since associations are binary, each association-end is declared together with its opposite association-end,

designated by the keyword **oppositeTo**. Multiplicities other than $*$ and **1** are specified using OCL invariants. Finally, comments are introduced with **//**.

As the following example illustrates, ComponentUML models specify how the application's data is structured, independently of how it will be visualized or accessed.

Example 1. We use a simple chatroom application as a running example throughout this paper. A demo version of this application can be found at [1]. The application provides an online discussion site where users converse by posting messages. Note that there are two types of users: registered and unregistered users. Registered users have their nicknames and passwords stored in the persistence layer. As usual, some options are only available to registered users, who log into the application by entering a valid nickname and password.

Here we use ComponentUML's textual syntax to model the chatroom's data model. The model, called ChatRoomDTM, consists of three entities representing chatrooms, registered users, and messages. The associations between these entities represent the relations between the registered users and the chatrooms in which they participate, the relations between the registered users and the messages that they have written, and the relations between the messages and the chatrooms where they have been posted. The entities' attributes represent that each chatroom has a topic, each chatroom can be public or not, each registered user has a nickname and a password, and each message has a body.

```
1  Entity Chatroom {
2    String topic
3    Boolean public
4    //registered users participating in this chatroom
5    Set(User) participants oppositeTo chatrooms
6    //messages posted in this chatroom
7    Set(Message) messages oppositeTo chatroom }

8  Entity User {
9    String nickname
10   String password
11   //chatrooms in which this registered user participates
12   Set(Chatroom) chatrooms oppositeTo participants
13   //messages written by this registered user
14   Set(Message) messages oppositeTo owner }

15 Entity Message {
16   String body
17   //chatroom where this message is posted
18   Chatroom chatroom oppositeTo messages
19   //registered user that wrote this message
20   User owner oppositeTo messages }
```

2.2 Object Constraint Language

The Object Constraint Language [8] is a language for specifying constraints and queries using a textual notation. As part of the UML standard, it was originally intended for modeling properties that could not be easily expressed using graphical notation, such as class invariants in a UML class diagram. Every OCL expression is written in the context of a model (called the *contextual model*), and is evaluated on an object model (also called the *instance* or *scenario*) of the contextual model. This evaluation returns a value but does not alter the given object model, since OCL's evaluation is side-effect free.

OCL is strongly typed. Expressions either have a primitive type, a class type, a tuple type, or a collection type. OCL provides standard operators on primitive data, tuples, and collections. For example, the operator →includes checks whether an object is part of a collection. OCL also provides a dot-operator to access the values of the objects' attributes and association-ends in the given scenario. For example, suppose that the contextual model includes a class c with an attribute at and an association-end as. Then, if o is an object of the class c in the given scenario, the expression $o.at$ refers to the value of the attribute at for the object o in this scenario, and $o.as$ refers to the objects linked to the object o through the association-end as. Finally, OCL provides operators to iterate over collections, such as →forAll, →exists, →select, →reject, →collect, and →iterate.

TABLE 1
SecureUML+ComponentUML: Actions and Resources

Resource	Atomic Actions	Composite Actions
Entity	create, delete	read, update, full access
Attribute	read, update	full access
Method	execute	
Association-end	read, create, delete	full access

2.3 SecureUML

SecureUML [6] extends role-based access control [9] with *authorization constraints*. These constraints can be used to specify policies that depend on properties of the system state, for example, that a user can only post a message to a chatroom where the user participates. More specifically, SecureUML allows one to formalize access control decisions that depend on two kinds of information:

1. *static information*, namely the assignments of users and permissions to roles, and the role hierarchy, and
2. *dynamic information*, namely the satisfaction of authorization constraints in the current system state.

SecureUML therefore supports the modeling of *roles* and their hierarchies, *permissions*, *actions*, *resources*, and *authorization constraints*. Moreover, one can also model assignments: which permissions are assigned to a role, which actions are allowed by a permission, which resources are affected by a permission, and which authorization constraint must be satisfied before granting a permission.

In our methodology, we use an extension of SecureUML to specify security policies over ComponentUML models. In this extension, the protected *resources* are the entities, along with their attributes, methods, and association-ends, while the *actions* are those shown in Table 1.

Note that there are two classes of actions: atomic and composite. *Atomic actions* are intended to map directly onto existing operations on the persistence layer. *Composite actions* hierarchically group lower-level actions. For example, the full access action for an attribute groups together the read and update actions for this attribute. Finally,

authorization constraints are specified using OCL, where the context of an OCL expression is the underlying ComponentUML model. Additionally, OCL expressions in SecureUML models may contain the variables self, caller, value, and target, which are interpreted as follows:

- self refers to the root resource upon which the action will be performed if the permission is granted. The root resource of an attribute, a method, or an association-end is the entity to which it belongs.
- caller refers to the user that will perform the action if the permission is granted.
- value refers to the value that will be used to update an attribute if the permission is granted.
- target refers to the object that will be added (or removed) at an association-end if the permission is granted.

The reader familiar with the original presentation of SecureUML [6] may notice that we have introduced two new variables that can be used in authorization constraints: the variables value and target. Furthermore, to avoid potential ambiguities, we have refined the association-end update action into two separate actions: association-end create and association-end delete.

In our concrete syntax, the entity modeling the system's users (or, more specifically, the system's *callers*) is declared with the keyword **User**. The roles that these users can take are declared with the keyword **Role** followed by the role's name, and its permissions, which are enclosed within brackets. The keyword **inherits**, appearing between two roles, declares that the first role is subordinated to the second role in the role hierarchy, and therefore inherits all its permissions.

Permissions are introduced by naming the root resources to which they grant access. Each permission consists of a list of actions through which the corresponding root resource can be accessed. Actions on attributes, methods, or association-ends are declared along with their names. For example, **Read::***attr* denotes the read action on the attribute *attr*. The **if-then** construction is used to declare that the permission to execute an action is constrained by a condition. This condition is the authorization constraint that is associated to the permission.

As the following example illustrates, SecureUML models specify the application's access control policy in a fine-grained way. These models depend, of course, on how the application's data is structured, but not on how it is visualized or accessed through the application's graphical user interface.

Example 2. We use the SecureUML's textual syntax to model a policy for posting and reading chatroom messages. Our model, called ChatRoomSTM, has two roles: the role DefaultR represents everybody, i.e., both registered and unregistered users, and the role UserR represents only registered users. Our policy states that everybody can read any message posted in a public chatroom, but that only registered users can read messages posted in a private chatroom, provided they participate in that chatroom. Moreover, only registered users can post messages in public chatrooms; they can also post to private chatrooms, provided they also participate in that chatroom.

```
1  User User
2  Role DefaultR {
3    Chatroom {
4      //everybody can access the messages posted in a
5      //public chatroom
6      if self.public then Read::messages }
7    Message {
8      //everybody can read the body of any message
9      //posted in a public chatroom
10     if self.chatroom.public then Read::body } }
11 Role UserR inherits DefaultR {
12   Chatroom {
13     //every registered user can access the messages that
14     //are posted in a chatroom in which she participates
15     if self.participants→includes(caller)
16       then Read::messages }
17   Message {
18     //every registered user can read the body of any
19     //message that is posted
20     //in a chatroom in which she participates
21     if self.chatroom.participants→includes(caller)
22       then Read::body
23     //every registered user can create a new message
24     Create
25     //every registered user can claim ownership of any
26     //unowned message
27     if self.owner.oclIsUndefined() and target=caller
28       then Create::owner
29     //every registered user can change the body of any
30     //message she owns
31     //provided it is not yet posted anywhere
32     if self.owner=caller
33         and self.chatroom.oclIsUndefined()
34       then Update::body
35     //every registered user can post in a public chatroom any
36     //message she owns,
37     //provided it is not yet posted anywhere
38     if self.owner=caller and target.public
39         and self.chatroom.oclIsUndefined()
40       then Create::chatroom
41     //every registered user can post, in a chatroom in which
42     //she participates, any message she owns,
43     //provided it is not yet posted anywhere
44     if self.owner=caller and target.participants
45         →includes(caller)
46       and self.chatroom.oclIsUndefined()
47       then Create::chatroom } }
```

SecureUML provides various constructs for expressing complex access control policies compactly and intuitively, for example, by using action and role hierarchies or by declaring default policies. Nevertheless, as described in Appendix A, every SecureUML model S can be uniquely transformed into a semantically equivalent model $S^\flat$ for which the following holds:

Remark 1. Let S be a SecureUML model. Then, for every atomic action act and every role r in S, there is exactly one permission in $S^\flat$ (possibly constrained by false) for r to execute act.

Informally, the model $S^\flat$ makes the security policy specified in S completely explicit. Thus, let *Auth* be the function that, for every SecureUML model S, role r, and action act, returns the authorization constraint associated to the unique permission that is defined in $S^\flat$ for r to execute act. We will use this function *Auth* to define the model-transformation that, in our methodology, lifts the security policy from the security model to the GUI model. We conclude this section with some examples that illustrate in which sense *Auth* makes the security policy specified in a security model explicit.

Example 3. Consider the chatroom's security model, **ChatRoomSTM**, in Example 2. Note that UserR is a subrole of DefaultR (in line 11), which means that UserR will inherit all the DefaultR's permissions. Thus, Auth(**ChatRoomSTM**, UserR, **Read::**body) returns:

self.chatroom.public (from ln. 10)
or
self.chatroom.participants→includes(caller) (from ln. 21).

Note also that the association-end messages is opposite to the association-end owner. This means that a create (respectively delete) action on messages will be constrained by the same authorization that constrains a create (respectively delete) action on owner, having simultaneously replaced the variable self by target and the variable target by self. Thus, although no permission is explicitly given for the role UserR to execute a create action on messages, Auth(**ChatRoomSTM**, UserR, **Create::**messages) returns:

target.owner.oclIsUndefined() and self=caller (from ln. 27).

Finally, note that there is no permission explicitly given to the role DefaultR for executing an action update on the attribute body. Since permissions are denied by default (and no other rules can be applied in this case, like the ones for role inheritance or opposite association-ends) Auth(**ChatRoomSTM**, UserR, **Update::**body) returns false.

3 GUI Models

GUI models provide a human-interface oriented view of a system. Together with data models, they constitute platform independent application models, omitting security aspects.

Informally, a GUI consists of widgets, which are visual elements that display information and trigger events that execute actions. In this section we present a key component of our methodology: a novel language for modeling GUIs for data-management applications, called GUIML (GUI Modeling Language). It is important, however, to understand that GUIML is a language for modeling not only the *structure* of a GUI, i.e., the elements (widgets) that comprise it, but also the GUI's *behavior*, i.e., how its elements will react (actions) in response to user interactions with them (events). In fact, the key feature of GUIML is the language it provides for modeling the GUI's behavior, which uses OCL to specify both the conditions and the arguments for the different actions. This feature enables both the security model and the GUI model to "speak" the same language (namely, OCL in the context of the common, underlying data model). This allows us to define rigorously the transformation function that lifts the security policy to the GUI level.

We next briefly describe the main elements of GUIML, namely, *widgets* (with their associated *variables*), *events*, and *actions*. We will also illustrate them later with a simple example: a window for our chatroom application, where users can read and post messages in a chatroom.

Widgets. A GUIML model consists of widgets of different types: windows (pages, when referring to web applications), combo-boxes (selectable lists), tables, date fields, boolean fields (check boxes), buttons, text fields, and labels. Widgets can be displayed in *containers*, which are also widgets. Widgets other than windows must be contained in another widget, and only windows, combo-boxes and tables may contain other widgets. Widgets may own variables, which store values for later use, and trigger events, which execute actions.

In concrete syntax, a widget is declared with a keyword like **Window**, **Button**, and **TextField**, according to its type, followed by the widget's (local) name, and the declaration of the variables it owns, the events it triggers, and the widgets it contains, all enclosed in brackets. The *global name* of each widget must be unique. If a widget is a window, its global name is the name given in its declaration. Otherwise, the global name results from concatenating, using dot, the global name of the widget's container with the name given in its declaration.

Variables. Each widget declaration may contain variable declarations, listing the variables owned by the widget. In concrete syntax, a variable declaration consists of the variable's type followed by its name.

There are also variables that are, by default, owned by every widget of a given type. These variables are implicitly declared in every widget declaration, and their values are handled in special ways. Here we only discuss the predefined variables that we will use in our example. The variables caller and role are predefined in every window. They store, respectively, the application's user and the user's role. The variable text is predefined in every label, button, and text field. This variable stores the string displayed on the screen within the label, button, and text field; also, when a user types in a text field, the value of its variable text is automatically updated. The variable rows is predefined in every combo-box and table. This variable stores the collection of items that can be selected from the combo-box or table. The variable row is also predefined in every combo-box and table where, for each row, it stores the item that can be selected.

Events. Each widget declaration may contain event declarations. Events are triggered when specific actions are executed upon their widgets, and they themselves can execute actions either on data or on other widgets.

The actions executed when an event is triggered are specified using *statements*. A statement is either an action, a conditional statement, an iteration, or a sequence of statements. In GUIML, the conditions in both conditional statements and iterations are specified using OCL expressions, whose context is the underlying ComponentUML model. Additionally, they can refer to the widget variables. In GUIML, when widget variables are used within OCL expressions,

they are enclosed in square brackets. Note that each sequence of statements associated to an event is executed as a single *transaction*: either all its statements successfully execute in the given order, or none of them are executed at all.

In concrete syntax, events are declared by indicating their types followed by the sequence of statements that they execute, enclosed in brackets. In our example we will use two types of events: **OnCreate** and **OnClick**. The former are triggered when the widgets are created and the latter are triggered when widgets are clicked upon. In particular, a window is created when an open action that has this window as its target is executed. All the other widgets are created immediately after their corresponding containers are created.

Actions. Every event declaration contains a sequence of statements that specifies the actions executed when the event is triggered. These actions can be executed either on objects belonging to the persistence layer or on objects belonging to the visualization layer. The former are called *data actions*, and the latter are called *GUI actions*. Note that some actions may take arguments whose values are only known at run-time, for example a delete action whose argument is the item selected by the user in a combo-box, or an update action whose argument is the number entered by the user in a text field. In GUIML, these values are specified using OCL. Again, the context of these expressions is the underlying ComponentUML model, but they can also refer to the widget variables.

Next, we briefly describe some of the GUIML data actions and their concrete syntax.

- *Entity create.* It creates a data item in the persistence layer. Its arguments are the *type* of the data item and the *variable* that stores the data item. It is declared by the statement *variable* **:= new** *type*.
- *Entity delete.* It deletes a data item from the persistence layer. Its argument is *object*, which is the data item deleted. It is declared by the statement **delete** *object*.
- *Attribute read.* It reads the value of a data item's attribute in the persistence layer. Its arguments are the data item *object* whose property is read, the *attribute* read, and the *variable* that stores the value read. It is declared by the statement *variable* **:=** *object*.*attribute*.
- *Attribute update.* It modifies the value of a data item's attribute in the persistence layer. Its arguments are the data item *object* whose attribute is modified, the *attribute* modified, and the new *value*. It is declared by the statement *object*.*attribute* **:=** *value*.
- *Association-end read.* It reads the collection of items linked to an item's association-end in the persistence layer. Its arguments are the data item *object* whose property is read, the association-end *assocEnd* read, and the *variable* that stores the collection read. It is declared by the statement *variable* **:=** *object*.*assocEnd*.
- *Association-end create.* It creates a link in the persistence layer between two data items. Its arguments are the source data item *srcObject*, the target data item *tgtObject*, and the association-end *assocEnd*-through which the target data item is linked to the source data item. An association-end create action is declared by the statement *srcObject*.*assocEnd* **+=** *tgtObject*.
- *Association-end delete.* It deletes a link in the persistence layer between two data items. Its arguments are the source data item *srcObject*, the target data item *tgtObject*, and the association-end *assocEnd* from which the target data item is removed. It is declared by the statement *srcObject*.*assocEnd* **−=** *tgtObject*.

Finally, we describe some of the GUI actions that are defined in GUIML.

- *Set.* It updates the value of a variable. Its arguments are the name of the *variable* and variable's new *value*. It is declared by the statement *variable* **:=** *value*.
- *Open.* It opens a window. Its argument is *target*, which names the window opened. Additionally, it may take as arguments any number of pairs $(variable_i, value_i)$, where $variable_i$ is the name of a variable owned by its *target* window, and $value_i$ is the value that is assigned to $variable_i$ when *target* is opened. It is declared by the statement **open** *target* **with** $variable_1$**:=** $value_1 \ldots variable_n$**:=** $value_n$.
- *Back.* It moves back to the previous window. It is declared using the keyword **back**.
- *Fail.* It forces a rollback in the current transaction, whereby the corresponding statement is not successfully executed. It is declared using the keyword **fail**.
- *Skip.* It does nothing. It is declared using the keyword **skip**.

We now provide an example that illustrates the main elements of the GUIML language: a window ReadPostWI for our chatroom application, where users can read and post messages in (previously selected) a chatroom. As this example will show, GUIML models depend on how the application's data is structured—after all, they describe how users interact with this data—but not on the application's access control policy.[1] In our example, this separation of concerns is reflected by the fact that the GUIML model for the window ReadPostWI is completely unaware of the security policy for reading and posting messages in our chatroom application.

Example 4. We use GUIML to model the window of our chatroom application where users can read and post messages in a chatroom. This window is named ReadPostWI. It owns a variable chatroomSel that stores a previously selected chatroom (the action that opens the window ReadPostWI will assign a value to this variable). The window ReadPostWI contains four widgets:

- a table ReadPostsTB for visualizing the messages posted in the selected chatroom;
- a text field WritePostEN for writing a new message;
- a button PostBU for posting in the selected chatroom the message written in the text field WritePostEN; and
- a button BackBU for moving back to the previous window.

1. Of course, in terms of the final application's *usability*, there is a dependency: an application's GUI can end up being unusable precisely because of the application's security policy.

```
1  Window ReadPostWI {
2    //this variable stores the previously selected chatroom
3    Chatroom chatroomSel
4    //this table visualizes the messages posted
5    //in the selected chatroom
6    Table ReadPostsTB {
7      OnCreate {
8        rows := [ReadPostWI.chatroomSel].messages } }
9    //in this text field the user writes its new message
10   TextField WritePostEN {
11     OnCreate { text := '' } }
12   //by clicking on this button, the user posts its new message
13   //in the selected chatroom
14   Button PostBU {
15     OnCreate { text := 'Post' } }
16   //by clicking on this button, the user moves back to
17   //the previous window
18   Button BackBU {
19     OnCreate { text := 'Back' } } }
20 //we continue with this table
21 Table ReadPostWI.ReadPostsTB {
22   //each column of this table shows the body of a message
23   //of the selected chatroom
24   Label BodyPostLB {
25     OnCreate {
26       text := [ReadPostWI.ReadPostTB.row].body } } }
27 //we continue with this button
28 Button ReadPostWI.PostBU {
29   OnClick {
30     newPost := new Message
31     newPost.owner += [ReadPostWI.caller]
32     newPost.body := [ReadPostWI.WritePostEN.text]
33     newPost.chatroom += [ReadPostWI.chatroomSel] } }
34 //we continue with this button
35 Button ReadPostWI.BackBU {
36   OnClick { back } }
```

Note that the table ReadPostTB and the buttons PostBU and BackBU are modeled partially inside the window ReadPostWI and partially outside this window. This is supported by our concrete syntax in order to improve the readability of the GUIML models. However, to avoid ambiguities, when a widget is modeled outside its widget container, the widget's global name is used. Note too that the table ReadPostTB is unaware of the security policy for visualizing messages, which in our running example states that only registered users are authorized to read messages posted in private chatrooms. Similarly, the button PostBU is unaware of the security policy for posting messages, which is that only registered users can post messages in public chatrooms and in private chatrooms but, in the latter case, they must also participate in these chatrooms.

4 Security-Aware GUI Models

In this section we describe the heart of our methodology: a model-transformation function *Sec* that, given a GUIML model G and a SecureUML model S, automatically generates a new GUIML model $\mathrm{Sec}(G, S)$. The generated model is identical to G except that it is *security aware* with respect to S. The transformation function *Sec* works by wrapping around every data action act in G an if-then-else statement with the following arguments:

- a condition that reflects the constraints associated to the permissions specified in S, for each of the different roles, to execute the action act;
- a then-branch that contains the action act; and
- an else-branch that contains the action **fail**.

Thus, the semantics of the if-then-else statement ensures that *act* will only be executed if the constraints associated to the corresponding permissions are satisfied. Moreover, this semantics also guarantees that, if these constraints are not satisfied, then the action **fail** will be executed, forcing a roll-back in the current transition.

More specifically, to generate the aforementioned if-then-else statement, the function *Sec* makes use of Remark 1. In particular, for each role r in S, it calls the function $Auth(S, r, act)$ to obtain the expression that ultimately (i.e., when the security policy is made completely explicit) constrains the permission given to r for executing act. However, since this expression may contain the variables self, value, target, and caller, the function *Sec* must also replace these variables by the actual arguments of the action act (including its actual user). We denote the resulting OCL expression by $\mathrm{Auth}(S, r, act)[args]$, where $args$ are the arguments specified in the GUI model for the action act. Finally, since different roles may be constrained by different expressions, the condition generated by *Sec* will have the form:

((r_1 = [$Window$.role] and Auth(S, r_1, act)[$args$])
or . . . or
(r_n = [$Window$.role] and Auth(S, r_n, act)[$args$])),

where $r_1, \ldots, r_n$ are all the roles declared in S. (Recall that the actual application's user and its role are always stored in the variables caller and role, which are owned by every window in the GUI model.)

The following examples illustrate the model-transformation function *Sec*. As previously mentioned, the complete, formal account of our methodology, including the model-transformation function *Sec*, is given in [7]. Nevertheless, the interested reader can find in Appendix B a high-level account of the correctness of *Sec*.

Example 5. Consider line 32 in Example 4. It specifies the third action that will be executed when the button ReadPostWI.PostBU is clicked upon, namely,

newPost.body := [ReadPostWI.WritePostEN.text].

Recall that **:=** refers to an update action, in this case to the action **Update::**body. The function *Sec* will replace this by the following if-then-else statement:

```
if ((DefaultR = [ReadPostWI.role] and false)
   or
   (UserR = [ReadPostWI.role] and
      ([newPost].owner = [ReadPostWI.caller]
      and [newPost].chatroom.oclIsUndefined())))
then newPost.body := [ReadPostWI.WritePostEN.text]
else fail.
```

To understand the condition generated by *Sec*, note that Auth(ChatRoomSTM, DefaultR, **Update::**body) is equal to false, but that Auth(ChatRoomSTM, UserR, **Update::**body) is equal to self.owner = caller and self.chatroom.oclIsUndefined(). Thus, the function *Sec* must replace the variable self by the newly created message newPost (since this is the object upon which the action **Update::**body will be executed), and the variable caller by ReadPostWI.caller (since this is the user that will execute the action **Update::**body).

Example 6. Consider line 31 in Example 4. It specifies the second action that will be executed when the button ReadPostWI.PostBU is clicked upon, namely,

```
newPost.owner += [ReadPostWI.caller].
```

Recall that **+=** refers to an association-end create action, in this case to the action **Create::**owner. Then, the function *Sec* will replace this line by the following if-then-else statement:

```
if ((DefaulR = [ReadPostWI.role] and false)
    or
    (UserR = [ReadPostWI.role] and
    ([newPost].owner.oclIsUndefined()
       and [ReadPostWI.caller]=[ReadPostWI.caller])))
then newPost.owner += [ReadPostWI.caller]
else fail.
```

To understand the condition generated by *Sec*, note that Auth(ChatRoomSTM, DefaultR, **Create::**owner) is equal to false, but that Auth(ChatRoomSTM, UserR, **Create::**owner) is equal to self.owner.oclIsUndefined() and target=caller. Thus, the function *Sec* must replace the variable self by the newly created message newPost (since this is the object upon which the action **Create::**owner will be executed), the variable caller by ReadPostWI.caller (since this is the user that will execute the action **Create::**owner), and the variable target also by ReadPostWI.caller (since the actual user is precisely the object that will be added by the **Create::**owner as the owner of the newly created message).

Our next example illustrates how our model transformation *Sec* leads to modularity and separation of concerns whereby the GUI model and the security model can be changed independently, if desired.

Example 7. Suppose that we decide to allow anyone (not only registered users, but also unregistered ones) to post messages in public chatrooms. To update the chatroom application's security-aware GUIML model, we just carry out the following steps:

- Step 1. Change the original chatroom's SecureUML model to reflect our security policy changes. We call the new security model PubChatRoomSTM and show below the new permissions for the role DefaultR (i.e., for everybody using the application) to create a message, update the body of a message, and post a message in a chatroom:

```
Role DefaultR {
  Message {
    //everybody can create a new message
    Create
    //everybody can change the body of
    //any unowned message
    //provided it is not yet posted anywhere
    if self.owner.oclIsUndefined()
       and self.chatroom.oclIsUndefined()
       then Update::body
    //everybody can post in a public chatroom
    //any unowned message
    //provided it is not yet posted anywhere
    if self.owner.oclIsUndefined() and target.public
       and self.chatroom.oclIsUndefined()
       then Create::chatroom } }
```

- Step 2. Apply our model transformation to the original chatroom GUIML model and the modified chatroom SecureUML model to generate the updated security-aware GUIML model. We show below the result of this transformation for line 32 in Example 4.

```
if ((DefaultR = [ReadPostWI.role] and
     ([newPost].owner.oclIsUndefined()
        and [newPost].chatroom.oclIsUndefined()))
    or
    (UserR = [ReadPostWI.role] and
     (([newPost].owner.oclIsUndefined()
        and [newPost].chatroom.oclIsUndefined())
     or
     ([newPost].owner = [ReadPostWI.caller]
        and [newPost].chatroom.oclIsUndefined()))))
then newPost.body := [ReadPostWI.WritePostEN.text]
else fail.
```

It is interesting to compare this result with the one explained in Example 5 for the case of the security model ChatRoomSTM. To understand the differences, note that Auth(ChatRoomSTM, DefaultR, **Update::**body) is equal to false, but that Auth(PubChatRoomSTM, DefaultR, **Update::**body) is equal to self.owner.oclIsUndefined() and self.chatroom.oclIsUndefined(). Also, recall that UserR inherits all permissions from DefaultR and, in particular, its new permission for updating the body of a message, which is constrained by Auth(PubChatRoomSTM, DefaultR, **Update**::body).

Note that in this example, the function *Sec* may generate conditions that can be further simplified. However, for the sake of illustration, here and elsewhere, we show the results of *Sec* without further simplification.

5 ActionGUI Toolkit and Applications

5.1 ActionGUI Toolkit

Security-aware GUIML models are platform independent and can be mapped to implementations employing different technologies. This includes desktop applications, web applications, and mobile applications. As part of our work, we built the ActionGUI Toolkit [1], which automatically generates web-based data-management applications from security-aware GUIML models.

The ActionGUI Toolkit features model editors for constructing and manipulating ComponentUML, SecureUML, and GUIML models. These editors share our own OCL parser, which takes as additional input the variables introduced by the different models, along with their respective types: in the case of SecureUML models, the variables self, caller, target, and value, and in the case of GUIML models, all the given widget variables. Crucially, the ActionGUI Toolkit implements our model transformation to generate security-aware GUIML models. Finally, it includes a code generator that, given a security-aware GUIML model, produces a web application based on the following, standard three-tier architecture.

1. *Presentation tier (also known as front-end).* Users access web applications through standard web browsers, which render the content (HTML and JavaScript) dynamically provided by the application server.

TABLE 2
Example Applications: Size of the Application's Models

	CRMApp	VMApp	MSMApp	eHRMApp	ChatApp
Widgets					
Number of windows	49	102	11	8	3
Number of buttons	182	293	30	18	10
Number of labels	691	697	83	66	7
Number of text fields	159	169	10	19	4
Number of boolean fields	67	9	0	5	1
Number of date fields	14	16	2	1	0
Number of combo boxes	52	33	24	1	0
Number of tables	65	85	7	9	2
Statements					
Number of if-then-else	650	334	150	35	7
Number of iterate	66	13	0	0	1
Data actions					
Number of creates (entity)	50	22	4	11	2
Number of deletes (entity)	14	33	0	0	2
Number of updates	268	180	15	25	4
Number of creates (assoc)	111	66	3	21	4
Number of deletes (assoc)	32	30	0	4	0
GUI actions					
Number of sets	1840	1553	569	120	24
Number of opens	164	234	18	7	7
OCL Expressions					
Number of expressions	3847	3221	925	331	74
Number of non-literal expressions	1478	1105	390	80	16

2. *Application tier.* The toolkit generates Java Web Applications, implemented using the Vaadin framework. The applications run in a servlet container (such as Tomcat or GlassFish), process client requests and, generate content, which is sent back to the client for rendering. They may also manipulate data stored in the persistence tier. When processing client requests, the generated application *interprets* its underlying security-aware GUIML model. In particular, it performs the required security checks before modifying any data stored in the persistence tier or sending any data to the presentation tier. This involves, of course, dynamically evaluating the OCL expressions appearing in the security-aware GUIML model.
3. *Persistence tier (also known as data tier or back-end).* The generated application manages information stored in a database. For each application, the toolkit generates the corresponding database schema from the application's ComponentUML model.

As a model-driven development tool, the ActionGUI Toolkit produces its best results when the data-management application's functionality can be reduced to CRUD actions and its dynamics consists of navigating and passing information through windows, and exchanging information with the underlying database. For applications in this category, ActionGUI automatically generates the complete implementation from the corresponding ComponentUML, SecureUML, and GUIML model. Note that calling CRUD actions is modeled in GUIML using data actions, and navigating and passing information through windows is modeled using GUI actions, namely, **open**, **back**, and **set**.

Of course, some data-management applications will require functionality that goes beyond CRUD actions. For example, they may need to send emails, print tables, or export data in some desired format. As expected, the ActionGUI Toolkit does not generate code for such methods. Instead, it includes their implementation—which must be provided by the application developer—in the generated application. When the application needs to interpret one of these methods, it simply calls the method provided.

5.2 Applications

We report here on five web applications that we developed using ActionGUI. Our objective is to show that one can use our methodology and the ActionGUI Toolkit to develop *non-toy* secure data-management applications. We begin by briefly describing our applications. In Table 2 we provide different measurements of the applications' size, defined in terms of their underlying GUIML models.

a) Customer Relationship Management (CRMApp). We have developed a web application for managing customers of a Hospital and Care Center. This application allows marketing and public relations personnel to manage contact information, including filtering contacts based on different criteria and exporting the results in Excel files. As customer data is highly sensitive, data is subject to a restrictive access-control policy. For example, a marketing and PR staff member can only access the contact information of those contacts previously selected as targets of a marketing campaign to which he is assigned. The application also allows a General Manager to create marketing campaigns, select the targeted patients, and assign marketing and PR staff members to campaigns.

b) Volunteer Management (VMApp). We have developed a web application for managing a care center's volunteer

program. Using this application, the program's coordinators can take actions such as: introduce new volunteers; create, edit, and modify tasks; and propose these tasks to the volunteers, based on the volunteers' time availability and preferences. The access-control policy stipulates, for example, that volunteers are only authorized to edit their own personal information, such as their preferences and time availability, and to accept or reject their own tasks.

c) Meal Service Management (MSMApp). This is a web application for managing a student residence's meal service. Using this application, a resident can notify the administration whether he will have a meal at the residence's cafeteria, in which of the available time slots, and if he will bring a guest. A resident shall only edit his own meal selection and within a specific time window, which depends on the selected meal. Administrators can create new resident accounts, and list the meals requested for each available time slot.

d) EHealth Record Management (eHRMApp). This is a web application for managing eHealth records. It allows users with the appropriate roles to: register new patients in a hospital and assign to them clinicians (doctor, nurses, etc.); retrieve patient information; register new nurses and doctors in a hospital and assign them to a ward; change nurses or doctors from one ward to another; and move patients to a different practice. The access-control policy regulates, in particular, access to the patients' highly sensitive records. These records shall only be retrieved by their handling doctors, although this policy can be relaxed in an emergency situation.

e) Chatroom (ChatApp). This is an extension of our running example: in addition to posting messages in a selected chatrooms, users can also create and delete chatrooms, under specific conditions.

CRMApp, VMApp, and MSMApp are commercial applications. They were developed for actual customers, and they are currently being used by their different stakeholders. In contrast, EHRMApp was developed as part of a case study proposed by industrial partners in a European project. The interested reader can find more information about this case study, as well as demo versions of the EHRMApp and ChatApp applications, at [1]. With respect to code-generation, for MSMApp, EHRMApp, and ChatApp, the ActionGUI's code generator automatically generates 100 percent of their implementation (no non-CRUD actions are ever called). In contrast, CRMApp and VMApp contain custom code for sending mails and for generating Excel files, which we borrowed from existing Java libraries. For all our examples, the ActionGUI's code generator produces the corresponding applications in under a minute.

We conclude this section by summarizing the key contributions of our methodology and toolkit. Our experience developing the reported applications provides evidence of the methodology's potential for developing real-world applications. First, the use of model-transformation and code generation frees the developer from programming fine-grained authorization constraints and inserting them at all the required places throughout the application's code and with the correct arguments. Except for small applications, this is cumbersome and error-prone, since the number of data actions associated to events may be on the order of hundreds; see, for example, the applications CRMApp and VMApp in Table 2. Second, our methodology supports modularity and separation of concerns. In particular, the security model can be changed independently of the GUI model, without requiring one to re-program and re-insert all the new fine-grained authorization constraints since this is automatically done by our model-transformation. This substantially aided developing our applications as our clients changed, several times, their security policies for CRMApp, VMApp, and MSMApp.

6 Related Work

Over the past 15 years, there have been numerous research advances in the model-driven development of data management applications. Among these, UWE [10], [11], [12], [13] and ZOOM [14] are the most closely related to our work.

As a modeling tool, UWE [10], [11], [12], [13] provides the modeler with a higher-level of abstraction than ActionGUI. In particular, the actions executed by the widgets' events are described in UWE using natural language. Thus, unless the models are appropriately refined, as discussed in [13], UWE does not support code-generation. In contrast, UWE provides specific diagrams for modeling GUI *presentations* and *navigations*, which facilitate the task of GUI modeling. In this respect, we define in [15] a mapping that transforms high-level UWE models into more concrete ActionGUI models that, once completed by the modeler, can be directly used to generate the intended applications. Finally, Busch [12] extends UWE to use SecureUML for modeling security policies. However, this work does not use a model-transformation to lift the security policy to the GUI level. Instead the UWE modeler is responsible for adding all the appropriate authorization checks to the GUI model.

Like ActionGUI, ZOOM [14] allows GUI modelers to specify widgets, their events, and their actions. Moreover, using an extension of Z [16], one can specify the conditions of the actions and their arguments, similar to how this is done in ActionGUI using OCL. In contrast to ActionGUI, ZOOM does not provide a language for modeling security and security aspects are not explicitly considered in this approach. Moreover, ZOOM does not support code-generation. It only provides interpreters for model animation.

In contrast to ActionGUI, UWE, and ZOOM, the approaches presented in [17], [18], [19] do not provide a language for modeling GUIs. They instead implement different *rules* for automatically deriving GUIs based on either the application's data model, as in [17], [18], or the application's prototypical scenarios, as in [19]. As expected, the behavior of the resulting GUIs is limited and, based on our experience, insufficient to cope with the logic embedded in real data-management applications. Moreover, security aspects are not addressed in these proposals.

There is other related work that falls between the two extremes of full GUI modeling and full GUI derivation. Both the OO-method [20], [21] and WebML [22], [23] support building GUIs using UI-*patterns*. These patterns specify the possible interactions with the application's data based on the classes, attributes, and associations that are declared in the underlying data model. These approaches have the advantage of reducing the time required for modeling

GUIs. However, the UI-patterns impose restrictions on the type of GUIs that can be modeled, both in terms of their structure and their behavior. Moreover, these approaches only support role-based access control, but not fine-grained access control. Other approaches that fall in this category are [24], [25]. In both cases, the modeler must associate to each widget container the specific data type accessed using the widget. As before, the possible interactions with the underlying data is limited by the default behavior implemented for these widget containers. Security aspects are also not considered.

Finally, there are approaches whose primary focus is to support UI design at different levels of abstraction. Prominent examples are the XML User Interface Language (XUL) [26] and the USer Interface eXtensible Markup Language (UsiXML) [27]. XUL is Mozilla's XML-based language for building user interfaces of applications like Firefox. UsiXML is an XML-compliant markup language that describes the UI for multiple usage contexts, such as Character User Interfaces (CUIs), Graphical User Interfaces, Auditory User Interfaces, and Multimodal User Interfaces.

Clearly, ActionGUI is designed for a different purpose than XUL and UsiXML. In particular, ActionGUI is designed for developing *secure* data-management applications. A key design decision for ActionGUI was to ensure that the security model and the GUI models "speak" the same language. To the best of our knowledge, neither XUL nor UsiXML are concerned with security aspects of the UIs. Moreover, ActionGUI is designed for the *model-driven* development of secure data-management applications and this has two clear consequences. First, ActionGUI's modeling languages are designed to be technology-agnostics, in contrast with XUL, which is tightly linked to Mozilla-related technologies. Second, ActionGUI's modeling languages are designed to support the automatic generation of ready to be deployed applications from the models. As a result, ActionGUI models are more concrete than general UsiXML models, which can be defined at any of the four abstraction levels specified in the Cameleon Reference Framework (CRF) [28]. In particular, a GUIML modeler always works at the CRF-Concrete UI level, while the WAR (Web application ARchive) file generated from a GUIML model (along with the associated security and data models) belongs to the CRF-Final UI level. Also, we note that [29] has carried out promising work on extending our methodology to cope with business processes, which are typically defined at the Task & Concepts abstraction level in CRF. Along these lines, it would be interesting to investigate ways of extending our methodology to support UI modeling at the CRF-Abstract UI level, where interaction details are abstracted away.

7 Conclusions

The methodology we proposed constitutes a further development of the idea of model-driven security [30]. The two main innovations are an expressive language for modeling an application's graphical user interface and behavior, and a many-models-to-model transformation that lifts a security policy specified on the application's data model to this behavioral model. Our transformation function captures the idea that authorization policies regulating complex transactions can be generated uniformly from much simpler policies on data. Despite our use of expressive modeling languages, we have shown for data-management applications that it is possible to generate automatically complete deployable applications.

Our methodology is supported by the ActionGUI Toolkit. Applications like those described in Section 5.2 show the toolkit's potential for developing real-world applications. Nevertheless, there is still much work ahead to turn this toolkit into a full, robust, industrial-strength development platform. In the short term, we plan to develop improved model editors and better support for integrating custom code. In the long term, we would like to support GUIs running on different platforms, like mobile devices. We also plan to add support for handling *privacy policies*: modeling and generating code to enforce that data usage must follow the purpose for which the data was collected and may entail obligations.

Finally, we would like to support model analysis, based on the formal semantics of our models and on the correctness of our model transformation. The following are examples of questions we would like to be able to formally answer. Will every sequence of action executed by every event in the model preserve the data model's invariants? Will authorization checks ever force a transaction roll-back? Do the conditions in the GUI model make redundant the authorization checks generated by the model transformation? Analysis support would allow us to optimize generated code and support assurance activities like system certification.

Appendix A

A.1 Making the Security Policy Explicit

In this appendix we define a transformation that, for every SecureUML model S, produces the SecureUML model $S^\flat$, which makes explicit the security policy declared in S. We define this transformation in four steps. Note that, as stated in Remark 1, the following holds at the end of our transformation: for every atomic action act and every role r in S, there is exactly one permission in $S^\flat$ (possibly constrained by false) for r to execute act.

Step 1. Copy the explicit permissions

- *Atomic actions.* Let act be an atomic action. Suppose that there is a permission in S for a role r to execute act under a constraint $auth$. Then, there is also a permission in $S^\flat$ for r to execute act under the same constraint $auth$.

Step 2. Unfold the security model

- *Action hierarchies.* Let CA be a composite action. Suppose that there is a permission in S for a role r to execute CA under a constraint $auth$. Then for every atomic action act contained in CA, there is a new permission in $S^\flat$ for r to execute act under the same constraint $auth$.
- *Role hierarchies.* Let act be an atomic action and let r and r' be two roles. Suppose that r is a subrole of r' in S, and that there is also a permission in S for r' to execute act under the constraint $auth$. There is then a

new permission in $S^\flat$ for the role r to execute act under the same constraint $auth$.

- *Delete actions.* Let $entity$ be an entity. Suppose that there is a permission in S for a role r to delete $entity$ under a constraint $auth$. Then for every association-end $assoc$ owned by $entity$, there is a new permission in $S^\flat$ for r to execute the action **Delete::**$assoc$ under the same constraint $auth$.
- *Opposite association-ends.* Let $assoc$ and $assoc'$ be two opposite association-ends. Let act be the action **Create::**$assoc$. Suppose that there is a permission in S for a role r to execute act under the constraint $auth$. There is then a new permission in $S^\flat$ for the role r to execute **Create::**$assoc'$ under the constraint that results from simultaneously replacing in $auth$ the variable self by target and the variable target by self. Unfolding is similar when act is the action **Delete::**$assoc$.

Step 3. Add default permissions to the security model

- *Denying by default.* Let r be a role and let act be an atomic action. Suppose that there is no permission in $S^\flat$ for the role r to execute act. There is then a new permission in $S^\flat$ for the role r to execute act under the constraint false. That is, the role r will be denied access to execute act in all circumstances.

Step 4. Simplify the resulting security model

- *Disjunction of constraints.* Let r be a role and let act be an action. Suppose that there are n permissions in $S^\flat$ for the role r to execute act. These n permissions are then simplified to a single permission whose authorization constraint results from disjoining together all the authorization constraints of the n individual permissions.

APPENDIX B

B. 1 Correctness of our Model Transformation

In this paper we have focused on our methodology and tool support for designing and generating secure data-management applications. Our approach also has a formal basis and we sketch here the correctness of our model transformation Sec, which is defined relative to the semantics of GUI models. Full details are provided in [7].

We define the semantics of GUI models by first giving a set of inference rules that defines a transition relation $\longrightarrow$ between triples of the form $\langle stm, I, \theta\rangle$, where stm is a statement, I is a scenario (i.e., an instance of the underlying data model), and θ represents a state of the widget variables. We provide inference rules for each possible statement: namely, for every type of data action and GUI action (base cases), and for arbitrary sequences of statements, conditional statements, and iterator-statements (inductive cases). In particular, for data actions, the inference rules have form

$$\overline{\langle act(args), I, \theta\rangle \longrightarrow \langle \mathbf{skip}, \mathrm{res}(I), \mathrm{res}(\theta)\rangle}\,,$$

where:

- $args$ are the arguments of the data action act,
- $\mathrm{res}(I)$ specifies the scenario that results from executing $act(args)$ in the scenario I and widget variable state θ, and
- $\mathrm{res}(\theta)$ specifies the widget variables' new state after executing $act(args)$ in the scenario I and widget variable state θ.

Crucially, no inference rule leading to **skip** is defined for the GUI action **fail**.

We then define the *operational semantics* of an event ev that executes the actions specified by an statement stm as the set of all the transitions

$$\langle stm, I, \theta\rangle \longrightarrow^* \langle \mathbf{skip}, I', \theta'\rangle\,,$$

where $\longrightarrow^*$ is the reflexive-transitive closure of $\longrightarrow$.

By definition, this operational semantics for events is *security-unaware*: it does not respect the authorization constraints that, according to the given security model, should constrain the execution of data actions. To provide a *security-aware* operational semantics for events, we define the *security-aware versions* of the inference rules. In particular, given a security model S, for each role r, and for each type of data action act, the security-aware version of the inference rule for act and r has the form

$$\frac{[\![(\mathrm{Auth}(S, r, act)[args])\theta]\!]^I = \mathrm{true}}{\langle act(args), I, \theta\rangle \longrightarrow \langle \mathbf{skip}, \mathrm{res}(I), \mathrm{res}(\theta)\rangle}\,,$$

where:

- $[\![expr]\!]^I$ denotes the value of the expression $expr$ in the scenario I; and, therefore,
- $[\![(\mathrm{Auth}(S, r, act)[args])\theta]\!]^I$ denotes the evaluation in the scenario I of the authorization $\mathrm{Auth}(S, r, act)$ that constrains the only permission that, according to Remark 1, ultimately allows users with the role r to execute the action act, given that arg are the arguments of act and θ is the state of the widget variables.

These security-aware inference rules define the transition relation $\longrightarrow_S$. Finally, given a security model S, we define the *security-aware operational semantics* of an event ev that executes the actions specified by an statement stm as the set of all the transitions

$$\langle stm, I, \theta\rangle \longrightarrow_S^* \langle \mathbf{skip}, I', \theta'\rangle\,.$$

The theorem below formalizes the *correctness* of our model-transformation function Sec. It states that evaluating a statement transformed by Sec following the security-unaware operational semantics returns the same result as evaluating the original statement using the *security-aware* semantics. Hence the transformed statement respects the authorization constraints formalized in the underlying security model.

Theorem. *Let S be a security model and let stm be a statement. Then, for every scenario I, and every widget variable state θ,*

$$\langle \mathrm{Sec}(stm, S), I, \theta\rangle \longrightarrow^* \langle \mathbf{skip}, I', \theta'\rangle \Longleftrightarrow \langle stm, I, \theta\rangle \longrightarrow_S^* \langle \mathbf{skip}, I', \theta'\rangle.$$

ACKNOWLEDGMENTS

This work was partially supported by the EU FP7-ICT Project "NESSoS: Network of Excellence on Engineering Secure Future Internet Software Services and Systems" (256980), by the Spanish Ministry of Science and Innovation Project "DESAFIOS-10" (TIN2009-14599-C03-01), by the Spanish Ministry of Economy and Competitiveness Project "StrongSoft" (TIN2012-39391-C04-04), and by Comunidad de Madrid Program "PROMETIDOS-CM" (S2009TIC-1465).

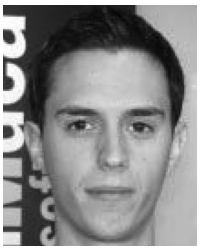
REFERENCES

[1] ActionGUI, "The ActionGUI project," http://www.actiongui.org, 2013.
[2] D.A. Basin, M. Clavel, M. Egea, M.A.G. de Dios, C. Dania, G. Ortiz, and J. Valdazo, "Model-Driven Development of Security-Aware GUIs for Data-Centric Applications," *Foundations of Security Analysis and Design VI*, pp. 101-124, Springer, 2011.
[3] D.A. Basin, M. Clavel, M. Egea, and M. Schläpfer, "Automatic Generation of Smart, Security-Aware GUI Models," *Proc. Second Int'l Symp. Eng. Secure Software and Systems (ESSoS '10)*, Feb. 2010, pp. 201-217.
[4] A. Kleppe, W. Bast, J.B. Warmer, and A. Watson, *MDA Explained: The Model Driven Architecture—Practice and Promise*. Addison-Wesley, 2003.
[5] Object Management Group, "Model Driven Architecture Guide v. 1.0.1," technical report, OMG, http://www.omg.org/cgi-bin/doc?omg/03-06-01, 2003.
[6] D. Basin, J. Doser, and T. Lodderstedt, "Model Driven Security: From UML Models to Access Control Infrastructures.," *ACM Trans. Software Eng. and Methodology*, vol. 15, no. 1, pp. 39-91, 2006.
[7] ActionGUI, "ActionGUI Semantics," technical report, IMDEA & ETH, http://www.actiongui.org, 2013.
[8] Object Management Group, "Object Constraint Language Specification Version 2.3.1," technical report, OMG, http://www.omg.org/spec/OCL/2.3.1, 2012.
[9] D.F. Ferraiolo, R.S. Sandhu, S. Gavrila, D.R. Kuhn, and R. Chandramouli, "Proposed NIST Standard for Role-Based Access Control," *ACM Trans. Information and System Security*, vol. 4, no. 3, pp. 224-274, 2001.
[10] H. Baumeister, N. Koch, and L. Mandel, "Towards a UML Extension for Hypermedia Design," *Proc. Second Int'l Conf. Unified Modeling Language: Beyond the Standard (UML'99)*, 1999, pp. 614-629.
[11] M. Busch and N. Koch, "MagicUWE—A Case Tool Plugin for Modeling Web Applications," *Proc. Ninth Int'l Conf. Web Eng. (ICWE '09)*, 2009, pp. 505-508.
[12] M. Busch, "Integration of Security Aspects in Web Engineering," master's thesis, Inst. für Informatik, Ludwig-Maximilians-Univ., 2011.
[13] C. Kroiss, N. Koch, and A. Knapp, "UWE4JSF: A Model-Driven Generation Approach for Web Applications," *Proc. Ninth Int'l Conf. Web Eng. (ICWE '09)*, 2009, pp. 493-496.
[14] X. Jia, A. Steele, L. Qin, H. Liu, and C. Jones, "Executable Visual Software Modeling—The ZOOM Approach," *Software Quality Control*, vol. 15, pp. 27-51, Mar. 2007.
[15] M. Busch and M.A.G. de Dios, "ActionUWE: Transformation of UWE to ActionGUI models," http://uwe.pst.ifi.lmu.de/publications/ActionUWE.pdf, 2012.
[16] J. Woodcock and J. Davies, *Using Z: Specification, Refinement, and Proof*. Prentice Hall, 1996.
[17] A.M.R. da Cruz, "Automatic Generation of User Interfaces from Rigorous Domain and Use Case Models," PhD dissertation, Faculdade de Engenharia da Univ. do Porto, Sept. 2010.
[18] A.M.R. da Cruz and J.P. Faria, "A Metamodel-Based Approach for Automatic User Interface Generation," *Proc. 13th Int'l Conf. Model Driven Eng. Languages and Systems: Part I (MODELS '10)*, Oct. 2010, pp. 256-270.
[19] M. Elkoutbi, I. Khriss, and R. Keller, "Automated Prototyping of User Interfaces Based on UML Scenarios," *Automated Software Eng.*, vol. 13, pp. 5-40, 2006.
[20] O. Pastor, J. Gómez, E. Insfrán, and V. Pelechano, "The OO-Method Approach for Information Systems Modeling: From Object-Oriented Conceptual Modeling to Automated Programming," *Information Systems*, vol. 26, no. 7, pp. 507-534, 2001.
[21] P.J. Molina, S. Meliá, and O. Pastor, "Just-UI: A User Interface Specification Model," *Proc. Int'l Conf. Computer-Aided Design of User Interfaces (CADUI '02)*, 2002, pp. 63-74.
[22] S. Ceri and P. Fraternali, "The Web Modeling Language—WebML," http://www.webml.org, 2003.
[23] Web Models Company, "Web Ratio—You Think, You Get," http://www.webratio.com, 2010.
[24] A. Schramm, A. Preußner, M. Heinrich, and L. Vogel, "Rapid UI Development for Enterprise Applications: Combining Manual and Model-Driven Techniques," *Proc. 13th Int'l Conf. Model Driven Eng. Languages and Systems: Part I*, Oct. 2010, pp. 271-285.
[25] V. Kulkarni, S. Reddy, and A. Rajbhoj, "Scaling Up Model Driven Engineering—Experience and Lessons Learnt," *Proc. 13th Int'l Conf. Model Driven Eng. Languages and Systems: Part II*, Oct. 2010, pp. 331-345.
[26] Mozilla Foundation, "XML User Interface Language (XUL)," https://developer.mozilla.org/en-US/docs/XUL, 2013.
[27] Q. Limbourg, J. Vanderdonckt, B. Michotte, L. Bouillon, and V. López-Jaquero, "UsiXML: A Language Supporting Multi-Path Development of User Interfaces," *Proc. Ninth IFIP Working Conf. Eng. for Human-Computer Interaction jointly with 11th Int'l Workshop Design*, 2004, pp. 200-220.
[28] G. Calvary, J. Coutaz, L. Bouillon, M. Florins, Q. Limbourg, L. Marucci, F. Paternò, C. Santoro, N. Souchon, D. Thevenin, and J. Vanderdonckt, "The CAMELEON Reference Framework," http://giove.isti.cnr.it/projects/cameleon.html, 2002.
[29] J. Valdazo, "Developing Secure Business Applications from Secure BPMN models," master's thesis, Facultad de Informática, Univ. Complutense de Madrid, 2012.
[30] D.A. Basin, M. Clavel, and M. Egea, "USenglish: A Decade of Model-Driven Security," *Proc. 16th ACM Symp. Access Control Models and Technologies (SACMAT '11)*, 2011, pp. 1-10.

David Basin received the PhD degree from Cornell University in 1989, and his Habilitation from the University of Saarbrücken in 1996. He is a full professor and has been the chair for Information Security in the Department of Computer Science, ETH Zurich, since 2003. From 2003 to 2011, he was the founding director of the ZISC, the Zurich Information Security Center. His research focuses on information security, in particular methods and tools for modeling, building, and validating secure and reliable systems.

Manuel Clavel received the bachelor's degree in philosophy from the Universidad de Navarra in 1992, and the PhD degree from the same university in 1998. He is currently an associate research professor at the IMDEA Software Institute and an associate professor at the Universidad Complutense de Madrid. His research focuses on rigorous, tool-supported model-driven software development, including: modeling languages, model transformation, model quality assurance, and code-generation.

Marina Egea received the PhD degree in computer science from the Universidad Complutense de Madrid in 2008. She has been a senior security consultant at Atos Research & Innovation, Madrid, since September 2011. From 2008 to 2011, she was a postdoctoral researcher, first with the Information Security Group at ETH Zurich, and later at IMDEA Software Institute. Her current research interests include secure systems development and assurance of security and privacy properties of cloud services.



Miguel A. García de Dios received the bachelor's and master's degrees from the Universidad Complutense de Madrid in 2007. He is currently working toward the PhD degree at IMDEA Software in the Modeling Group. His research focuses on rigorous, tool-supported model-driven software development, including: modeling languages, model transformation, model quality assurance, and code generation.

Carolina Dania received the bachelor's degree from the Universidad Nacional de Córdoba, Argentina, and the master's degree from the Universidad Complutense de Madrid, Spain. She is currently working toward the PhD degree at IMDEA Software in the Modeling Group. Her research interests include software engineering, formal methods, and security. In particular, she is working in tools and techniques for modeling, building and validating secure and reliable software systems.

Model-driven Development of a Secure eHealth Application

Miguel A. García de Dios[1], Carolina Dania[1], David Basin[2], and Manuel Clavel[1]

[1] IMDEA Software Institute, Madrid, Spain
[miguelangel.garcia,carolina.dania,manuel.clavel]@imdea.org
[2] ETH Zürich, Switzerland
basin@inf.ethz.ch

Abstract. We report on our use of ActionGUI to develop a secure eHealth application based on the NESSoS eHealth case study. ActionGUI is a novel model-driven methodology with an associated tool for developing secure data-management applications with three distinguishing features. First, it enables a model-based separation of concerns, where behavior and security are modeled individually and subsequently combined. Second, it supports model-based quality assurance checks, where the properties proven about the models transfer to the generated applications. Finally, for data-management applications, the ActionGUI tool automatically generates complete, ready-to-deploy, security-aware, web applications. We explain these features in the context of the eHealth application.

1 Introduction

In [3] we proposed a novel methodology, called ActionGUI, for the model-driven development of secure data-management applications. This methodology enables a model-based separation of concerns, where an application's behavior and security are modeled individually and subsequently combined. Moreover, it supports model-based quality assurance checks, where relevant properties may be proven about the combined models. These properties then transfer to the automatically generated data-management applications.

We report here on our use of ActionGUI to develop a secure data-management application. This application is based on a case study proposed within NESSoS, the European Network of Excellence on Engineering Secure Future Internet Software Services and Systems [12]. The eHealth case study consists of a web-based system for electronic health record management (EHRM). Electronic health records (EHR) store information created by, or on behalf of, a health professional in the context of the care of a patient.

Electronic health records are highly sensitive and therefore their access must be controlled. Part of the challenge in this case study was to model the access control policy and build an application that enforces it at runtime. The policy consists of various authorization rules along the lines of: *The access control criteria for an EHR depends, among others, on the type of EHR. For instance,*

a highly sensitive record might be only available to the patient's treating doctor (and perhaps a few others, in rare situations). Such rules necessitate fine-grained access control, where access control decisions depend not only on the user's credentials but also on the satisfaction of constraints on the state of the persistence layer, i.e. on the values of stored data items.

We show how ActionGUI's modeling languages can be used to specify the application's data model (e.g., hospital staff, health records), security policy (e.g., rules like the above) and behavior. Moreover, by illustrative examples, we highlight various features of the ActionGUI methodology and associated tool. Overall, the eHealth case study is interesting as an example of developing a secure data-management application and it provides a proof-of-concept for the application of the ActionGUI methodology to an industry-relevant problem.

Organization. In Section 2 we provide background on the ActionGUI methodology and tool. In Section 3 we give an account of our modeling and generation of the EHRM application with ActionGUI. In Section 4 we describe a proof method for checking that the behavior of the modeled data-management application respects the invariants of the application's underlying data model, and we apply it to our EHRM models. Finally, in Section 5, we draw conclusions.

2 ActionGUI

ActionGUI [3] is a methodology for the model-driven development of secure data-management applications. It consists of languages for modeling multi-tier systems, and a toolkit for generating these systems. Within this methodology, a secure data-management application is modeled using three interrelated models:

1. A *data model* defines the application's data domain in terms of its classes, attributes, associations, and methods.
2. A *security model* defines the application's security policy in terms of authorized access to the actions on the resources provided by the data model.
3. A graphical user interface, or *GUI model*, defines the application's graphical interface and application logic. Note, in particular, that this model formalizes both *UI structure* and *behavior*.

The heart of this methodology, illustrated in Figure 1, is a model-transformation function that automatically lifts the policy that is specified in the security model to the GUI model. The idea is simple but powerful. The security model specifies under what conditions actions on data are authorized. The control information in the GUI model specifies which actions are executed in response to which events. Lifting essentially consists of prefixing each data action in the GUI model with the authorization check specified in the security model. The resulting GUI model is security aware. It specifies UI structure, information flow with persistent storage, and all authorization checks.

The ActionGUI methodology is implemented within a toolkit, also called ActionGUI [1], which performs the aforementioned many-models-to-model transformation. From the resulting security-aware GUI model, ActionGUI generates

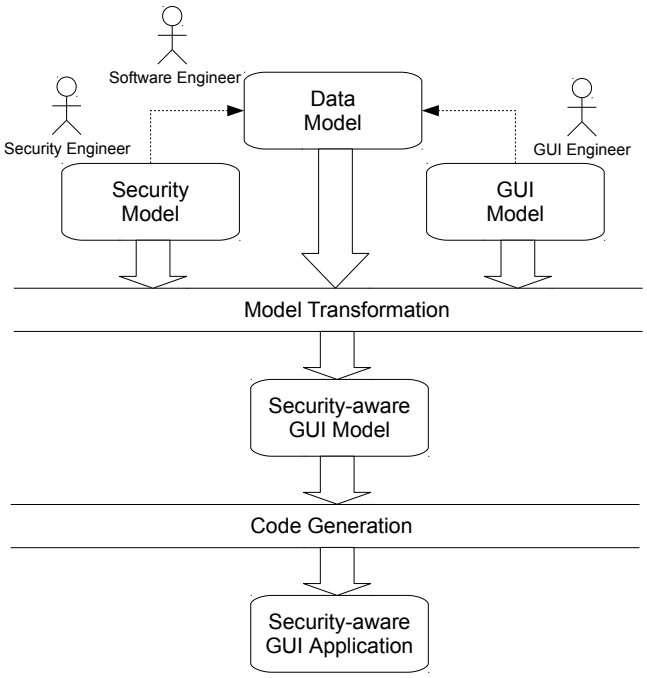


Fig. 1. Model-driven development of security-aware GUIs.

a deployable application along with all support for access control. In particular, when the security-aware GUI model contains only calls to execute CRUD actions, i.e., those actions that create, read, update, and delete data, then ActionGUI will generate the complete implementation automatically.

In the remaining part of this section we briefly introduce the languages that are used within the ActionGUI methodology to model the applications' data, security, and GUI models, including their constraints, as well as the tools supporting the ActionGUI methodology. In the next section we will use the NESSoS EHRM application scenario to illustrate these modeling languages as well as the model-based separation of concerns supported by the ActionGUI methodology.

2.1 Data models

Data models provide a data-oriented view of a system. They typically specify how data is structured, the format of data items, and their logical organization, i.e., how data items are grouped and related. ActionGUI employs ComponentUML [4] for data modeling. ComponentUML provides a subset of UML class models where *entities* (classes) can be related by *associations* and may have *attributes* and *methods*.

2.2 Constraints

The Object Constraint Language (OCL) [13] is a language for specifying constraints and queries using a textual notation. ActionGUI supports different uses of OCL: it is used in data models to specify *data invariants*, in security models to specify *authorization constraints*, and in GUI models to specify if-then-else *conditions* and action *arguments*.

Every OCL expression is written in the context of a model (called the *contextual model*), and is evaluated on an object model (also called the *instance* or *scenario*) of the contextual model. This evaluation returns a value but does not alter the given object model, since OCL's evaluation is side-effect free.

OCL is strongly typed. Expressions either have a primitive type, a class type, a tuple type, or a collection type. OCL provides: standard operators on primitive data, tuples, and collections; a dot-operator to access the values of the objects' attributes and association-ends in the given scenario; and operators to iterate over collections. Particularly relevant for its use in ActionGUI models, OCL includes two constants, null and invalid, to represent undefinedness. Intuitively, null represents unknown or undefined values, whereas invalid represents error and exceptions. To check if a value is null or invalid, OCL provides, respectively, the Boolean operators oclIsUndefined() and oclIsInvalid().

2.3 Security models

SecureUML [4] extends Role-Based Access Control (RBAC) [9] with *authorization constraints*. These constraints are used to specify policies that depend on properties of the system state. SecureUML supports the modeling of *roles* and their hierarchies, *permissions*, *actions*, *resources*, and *authorization constraints*.

In ActionGUI, we use an extension of SecureUML for specifying security policies over data models. In this extension:

- The protected *resources* are the entities, along with their attributes, methods, and association-ends.
- The controlled *actions* are: to create and delete entities; to read and update attributes; to read, create, and delete association-ends; and to execute methods.
- The authorization constraints are specified using OCL.

The contextual model of the authorization constraints is the underlying data model. Additionally, authorization constraints may contain the variables self, caller, value, and target, which are interpreted as follows:

- self refers to the root resource upon which the action will be performed if permission is granted. The root resource of an attribute, a method, or an association-end is the entity to which it belongs.
- caller refers to the user that will perform the action if the permission is granted.
- value refers to the value that will be used to update an attribute if the permission is granted.

– target refers to the object that will be added to (or removed from) the (root) resource at an association-end if the permission is granted.

2.4 GUI models

GUI models provide a human-interface oriented view of a system. A GUI consists of widgets, which are visual elements that display information and trigger events that execute actions. In ActionGUI, we use GUIML [3] for modeling both

– the GUI's *structure*, i.e, the elements (*widgets*) that comprise it,
– and the GUI's *behavior*, i.e., how its elements will react (*actions*) in response to user interactions with them (*events*).

Behavioral modeling is a key feature of GUIML and uses OCL to specify both the conditions and the arguments for the different actions; the contextual model of these conditions and arguments is again the underlying data model. This enables both the security model and the GUI model to "speak" the same language, namely OCL in the context of the common, underlying data model. This allows us to define rigorously the transformation function that lifts the security policy to the GUI level.

We next briefly describe the main elements of GUIML, namely, *widgets* (with their associated *variables*), *events*, and *actions*.

Widgets. A GUI model consists of widgets of different kinds. Examples include windows (pages, when referring to web applications), combo-boxes (selectable lists), tables, date fields, boolean fields (check boxes), buttons, text fields, and labels.

Variables. Widgets may own *variables*, which store values for later use. Each widget declaration may contain variable declarations, listing the variables owned by the widget. There are variables that are, by default, owned by every widget of a given type. In particular, the variables caller and role are predefined in every window. They store, respectively, the application's user and the user's role.[3] The variable text is predefined in every label, button, and text field. This variable stores the string displayed on the screen within the label, button, and text field. The variable rows is predefined in every combo-box and table. This variable stores the collection of items that can be selected from the combo-box or table. The variable row is also predefined in every combo-box and table where, for each row, it stores the item that corresponds to this row. Finally, the variable selected is also predefined in every combo-box or table where it stores the item(s) selected in the combo-box or table.

[3] Currently it is a task for the GUI modeler to guarantee that the variables caller and role always store an *authenticated* user and a valid role. This can be done, for example, by modeling a login window where the users will need to enter a valid nickname and password before accessing the application.

Events. Widgets may trigger events, which execute actions either on data or on other widgets. The actions executed when an event is triggered are specified using *statements*. A statement is either an action, a conditional statement, an iteration, a try-catch, or a sequence of statements. The conditions in conditional statements are specified using OCL expressions, whose context is the underlying data model. Additionally, they can refer to the widget variables. Note that each sequence of statements associated to an event is executed as a single *transaction*: either all statements in the sequence successfully execute in the given order, or none of them are executed at all.

Actions. Events trigger actions that can be executed either on objects belonging to the persistence tier or on objects belonging to the presentation tier. The former are called *data actions* and the latter are called *GUI actions*. Data actions are precisely those controlled in the security model, namely: to create and delete entities; to read and update attributes; to read, create, and delete association-ends; and to execute methods. GUI actions include those for setting the value of a widget variable, opening a window (**open**), moving back to the previous window (**back**), and forcing a rollback of the current transaction (**fail**). Note that some actions may take arguments. The values of these arguments are specified using OCL expressions, whose context is the underlying data model, and they can also refer to the widget variables.

2.5 Security-aware GUI models

The heart of ActionGUI is a model-transformation function *Sec* that, given a GUIML model G and a SecureUML model S, automatically generates a new GUIML model $\mathrm{Sec}(G, S)$. The generated model is identical to G except that it is *security aware* with respect to S. The transformation function *Sec* works by wrapping around every data action *act* in G an if-then-else statement with the following arguments:

- a condition that reflects the constraints associated to the permissions specified in S, for each of the different roles, to execute the action *act*;
- a **then** branch that contains the action *act*; and
- an **else** branch that contains the action **fail**.

Thus, the semantics of an if-then-else statement ensures that *act* will only be executed if the constraints associated to the corresponding permissions are satisfied. Moreover, if these constraints are not satisfied, then the action **fail** will be executed, forcing a rollback in the current transaction.

2.6 Tool support

Security-aware GUI models are platform independent and can be mapped to implementations employing different technologies. This includes desktop applications, web applications, and mobile applications. The ActionGUI Toolkit [1],

automatically generates web-based data-management applications from security-aware GUIML models.

The ActionGUI Toolkit features model editors for constructing and manipulating ComponentUML, SecureUML, and GUIML models. Crucially, the ActionGUI Toolkit implements our model transformation to generate security-aware GUIML models. Moreover, it includes a code generator that, given a security-aware GUIML model, produces a web application based on the following three-tier architecture:

1. Presentation tier (also known as front-end): Users access web applications through standard web browsers, which render the content (HTML and JavaScript) dynamically provided by the application server.
2. Application tier: The toolkit generates Java Web Applications, implemented using the Vaadin framework. The applications run in a servlet container (such as Tomcat or GlassFish), process client requests, and generate content, which is sent back to the client for rendering.
3. Persistence tier (also known as data tier or back-end): The generated application manages information stored in a database.

3 The EHRM ActionGUI Application

The NESSoS EHRM application scenario defines different system use cases along with the associated access control policy. The use cases include: register new patients in a hospital and assign them to clinicians, such as nurses or doctors; retrieve patient information; register new nurses and doctors in a hospital and assign them to a ward; change nurses or doctors from one ward to another; and reassign patients to doctors. Due to space limitations, we will not describe how we model all of these use cases. We focus instead on a representative use case as a running example: reassigning patients to doctors. We will use this example to illustrate ActionGUI's modeling languages as well as the model-based separation of concerns supported by the ActionGUI methodology.

3.1 The EHRM's data model

The full data model for the EHRM application contains 18 entities, 40 attributes, and 48 association-ends. We discuss below just the entities, attributes, and association-ends that are required for our running example.

Figure 2 presents this data model, formalized using ActionGUI's textual syntax. In this syntax, entities are declared with the keyword **entity** followed by the entity's name, and its attributes and association-ends, which are enclosed within brackets. Attributes and association-ends are declared together with their types. Moreover, since associations are binary, each association-end is declared together with its opposite association-end, designated by the keyword **oppositeTo**.

As this example shows, ActionGUI data models specify how the application's data is structured, independently of how it will be visualized or accessed.

Professional. This entity represents the EHRM's users. The role assigned to each user is specified by its role attribute. The roles considered are DIRECTOR, ADMINISTRATOR, DOCTOR, NURSE, and SYSTEM. The medical centers where a user works are linked to the user through the association-end worksIn. If a user is a doctor, then it is linked to the corresponding doctor information through the association-end asDoctor. Similarly, if a user is an administrative staff, then it is linked to staff information through the association-end asAdministrative.

MedicalCenter. This entity represents medical centers. The departments belonging to a medical center are linked to the medical center through the association-end departments. The professionals working for a medical center are linked to the medical center through the association-end employees.

Doctor. This entity represents doctor information. Doctor information is linked to the corresponding professional through the association-end doctorProfessional. The departments where a doctor works are linked to the doctor's information through the association-end doctorDepartments. The patients treated by a doctor are linked to the doctor's information through the association-end doctorPatients.

Administrative. This entity represents administrative staff information. Administrative staff information is linked to the corresponding professional through the association-end administrativeProfessional.

Department. This entity represents departments. The medical center to which a department belongs is linked to the department through the association-end belongsTo. The doctors working in a department are linked to the department through the association-end doctors. The patients treated in a department are linked to the department through the association-end patients.

Patient. This entity represents patients. The doctor treating a patient is linked to the patient through the association-end doctor. The department where a patient is treated is linked to the patient through the association-end department.

3.2 The EHRM data model's invariants

The full EHRM application data model is constrained by 66 data invariants, formalized using OCL. The following three invariants are representative.

1. *Each patient is treated by a doctor.*
   Patient.allInstances()→forAll(p|not(p.doctor.oclIsUndefined()))
2. *Each patient is treated in a department.*
   Patient.allInstances()→forAll(p|not(p.department.oclIsUndefined()))
3. *Each patient is treated by a doctor who works for a set of departments, including the department where the patient is treated.*
   Patient.allInstances()→forAll(p| p.doctor.doctorDepartments→includes(p.department))

```
entity Professional {
  Role role
  Set(MedicalCenter) worksIn oppositeTo employees
  Doctor asDoctor oppositeTo doctorProfessional
  Administrative asAdministrative oppositeTo administrativeProfessional  }
entity MedicalCenter {
  Set(Department) departments oppositeTo belongsTo
  Set(Professional) employees oppositeTo worksIn  }
entity Doctor {
  Professional doctorProfessional oppositeTo asDoctor
  Set(Department) doctorDepartments oppositeTo doctors
  Set(Patient) doctorPatients oppositeTo doctor  }
entity Administrative {
  Professional administrativeProfessional oppositeTo asAdministrative  }
entity Department {
  MedicalCenter belongsTo oppositeTo departments
  Set(Doctor) doctors oppositeTo doctorDepartments
  Set(Patient) patients oppositeTo department  }
entity Patient {
  Doctor doctor oppositeTo doctorPatients
  Department department oppositeTo patients  }
enum Role { DIRECTOR ADMINISTRATOR DOCTOR NURSE SYSTEM }
```

Fig. 2. The eHRMApp's data model (partial).

These invariants make precise the intended meaning of the associations between the entities Patient, Doctor, and Department. The first two invariants state that the doctor and the department associated to a patient cannot be undefined, i.e., *null*. The third invariant states that a doctor who treats a patient must work in the department where the patient is treated, although the doctor may also work in other departments.

3.3 The EHRM's security model

Electronic health records are by their nature highly sensitive and the NESSoS case study informally defines the policy that regulates their access. As expected, the authorization to carry out certain actions is not only role-based, but also context-based. In other words, the EHRM access control policy is *fine grained*.

The full EHRM application's security model contains 5 roles and 573 permissions, where each permission authorizes users in a role to execute an action upon the satisfaction of an authorization constraint formalized in OCL. In Figure 3 we present examples of two permissions, modeled using ActionGUI's textual syntax. In this syntax, the roles that users can take are declared with the keyword **role** followed by the role's name, and its permissions, which are enclosed within brackets. Permissions are introduced by naming the root resources to which they grant access. Each permission consists of a list of actions through which the cor-

responding root resource can be accessed. Actions on attributes, methods, or association-ends are declared along with their names. For example, **update(***attr***)** denotes the update action on the attribute *attr*. The keyword **constrainedBy** is used to declare that the permission to execute an action is constrained by the given condition (enclosed in square brackets).

The first permission authorizes a user (*caller*) with the role ADMINISTRATOR to reassign a patient to a department (*value*) provided that the user works in a set of medical centers that includes the one to which the department belongs where the patient will be reassigned. The second permission authorizes a user (*caller*) with the role ADMINISTRATOR to reassign a patient (*self*) to a doctor (*value*) provided two conditions are satisfied: (i) among the medical centers where the user works, there is at least one where the doctor to which the patient will be reassigned also works; and (ii) the user works in medical centers that includes the center to which the department belongs where the patient is currently being treated. Note that no other role has permissions associated to the actions of reassigning a patient to a department or to a doctor.

As this example illustrates, ActionGUI security models are formulated in terms of the application's data. This formalization is independent of how the data is visualized or accessed through the application's graphical user interface.

```
1  role ADMINISTRATOR {
2    Patient{
3      update (department) constrainedBy [caller.worksIn→includes(value.belongsTo)] }
4    Patient{
5      update (doctor) constrainedBy
6        [caller.worksIn→exists(m|value.doctorProfessional.worksIn→includes(m))
7          and caller.worksIn→includes(self.department.belongsTo)] }
```

Fig. 3. Examples of the EHRM security model's permissions.

3.4 The EHRM's GUI model

The full EHRM application's GUI model contains 8 windows for the following use cases: login to the application; access a medical center's information; register a new patient; review a patient's information; reassign a patient to a doctor and department; access options reserved for the medical center's director; introduce a professional into the system; and reassign a professional to a department.[4]

[4] Here are some other concrete figures about the size of the GUI model: i) Widgets: 19 buttons; 73 labels; 19 text fields; 5 boolean fields; 1 date field; 1 combo-box; and 9 tables; ii) Statements: 34 if-then-else statements; iii) Data actions: 11 create actions; 41 update actions; 5 add link actions; and 2 remove link actions; iv) GUI actions: 157 set actions; and 7 open actions; v) OCL expressions: 361 expressions (77 non-literals).

We discuss below the window relevant for our running example: the window movePatientWI for reassigning a patient to a doctor and a department. Figures 4 and 5 present our model of this window, in ActionGUI's textual syntax. Figure 6 contains a screenshot of the actual window generated from this model.

In ActionGUI's textual syntax, a widget is declared with a keyword like **Window**, **Button**, and **TextField**, according to its type, followed by the widget's name, and the declaration of the variables it owns, the events it triggers, and the widgets it contains, all enclosed in brackets. A variable declaration consists of the variable's type followed by its name, possibly followed by the variable's initial assignment (if any) and by the statement that will be executed every time the variable's value changes (if any), the latter enclosed in brackets. Events are declared by indicating their types followed by the sequence of statements that they execute, enclosed in brackets. The syntax for declaring the different data and GUI actions should be clear from the example below.

The window movePatientWI assumes that both a medical center and a patient have previously been selected. This information is stored, respectively, in the variables medicalCenter and patient (lines 2-3). The window movePatientWI contains the following widgets:

- A label patientLa that displays the name and surname of the selected patient (lines 5–7).
- A label departmentLa that displays the name of the department where the selected patient is treated (lines 8–9).
- A label doctorLa that displays the name and surname of the doctor who treats the selected patient (lines 10–13).
- A label departmentsLa that displays a message inviting the user to select a department (lines 14–15).
- A label doctorsLa that displays a message inviting the user to select a doctor (lines 16–17).
- A table departmentsTa that displays information about the departments that belong to the selected medical center (line 22); in particular, the name of each of these departments is shown (line 31-34). Also, when the user selects a department from this list, it refreshes the list of doctors displayed in the table doctorsTa (see below) with the doctors who work for the selected department (lines 19–21).
- A table doctorsTA that is initially empty (line 24). As previously explained, upon selection of a department in the table departmentsTa, it displays information about the doctors who work for the selected department (lines 19–21); in particular, the name and surname of each of these doctors are shown (lines 35-41).
- A button moveBu that, when clicked upon, if there is a department selected in the table departmentsTa (line 44), and there is also a doctor selected in the table doctorsTa (line 45), then:
  - it reassigns the selected department to the selected patient (line 46);
  - it reassigns the selected doctor to the selected patient (line 47);
  - it notifies the user that the reassignment succeeded (lines 48).

Otherwise, it notifies the user that either a doctor (line 50) or a department (line 52) must first be selected.
- A button backBU that, when the user clicks on it, it returns to the previous window (line 55).

As this example illustrates, ActionGUI GUI models depend on how the application's data is structured — after all, they describe how users interact with this data — but not on the application's security policy. Of course, in terms of the final application's *usability*, there is a dependency: a GUI can end up being unusable precisely because of the application's security policy.

3.5 The EHRM's security-aware GUI model

As explained in Section 2.5, the heart of ActionGUI is a model-transformation function that, essentially, prefixes each data action in the GUI model with the authorization check specified in the security model. The full EHRM application's GUI model contains 59 data actions, and therefore the automatically generated EHRM application's security-aware GUI model contains the same number of authorization checks.

To illustrate our model-transformation function, we show in Figure 7 the part of the security-aware GUI model for the button moveBu's event **onClick** that is relevant for our running example. The action of reassigning the selected patient to the department selected in the table departmentsTa (line 46 in Figure 5) is now wrapped by an if-then-else statement (lines 46.1-46.5 in Figure 7) whose condition reflects the permission for executing this action given by line 3 in Figure 3. Similarly, the action of reassigning the selected patient to the doctor selected in the table doctorsTa (line 47 in Figure 5) is wrapped by an if-then-else statement (lines 47.1-47.7 in Figure 7) whose condition reflects the permission for executing this action given by lines 5–7 in Figure 3.

3.6 Generating the EHRM application

The ActionGUI Toolkit automatically generates the complete EHRM application in under 10 seconds. The generated .war file includes the Vaadin library as well as other external libraries. The Vaadin library is responsible of 70% of the size of the generated file and only 10% of this file corresponds to the code that ActionGUI automatically generates to interpret the application's model. The size of the .war file containing the complete application is roughly 15 MB.

4 Analyzing the EHRM ActionGUI Application

Model-Driven Architecture supports the development of complex software systems by generating software from models. Of course, the quality of the generated code depends on the quality of the source models. If the models do not properly

```
1  Window movePatientWi {
2    MedicalCenter medicalCenter
3    Patient patient
4    String text := ['Move a patient']

5  Label patientLa {
6    String text := ['Patient: '.concat($movePatientWi.patient$.contact.name)
7                    .concat(' ').concat($movePatientWi.patient$.contact.surname)] }
8  Label departmentLa {
9    String text := ['Department: '.concat($movePatientWi.patient$.department.name)] }

10 Label doctorLa {
11   String text := ['Doctor: '.concat($movePatientWi.patient$.doctor.
12                   doctorProfessional.name).concat(' ').
13                   concat($movePatientWi.patient$.doctor.doctorProfessional.surname)] }

14 Label departmentsLa {
15   String text := ['Select the new department:'] }

16 Label doctorsLa {
17   String text := ['Select the new doctor:'] }

18 Table departmentsTa {
19   Department selected {
20     if [not $selected$.oclIsUndefined()] {
21       movePatientWi.doctorsTa.rows := [$selected$.doctors] } }
22   Set(Department) rows := [$movePatientWi.medicalCenter$.departments] }

23 Table doctorsTa {
24   Set(Doctor) rows := [Doctor.allInstances()→select(false)]
25   Doctor selected }

26 Button moveBu {
27   String text := ['Move the patient'] }

28 Button backBu {
29   String text := ['Back']
30 }
```

Fig. 4. A window for reassigning a selected patient (part I)

```
31 Table movePatientWi.departmentsTa {
32   columns{
33           ['Department'] : Label department {
34                               String text := [$departmentsTa.row$.name] } } }

35 Table movePatientWi.doctorsTa {
36   columns {
37           ['Doctor'] : Label doctor {
38                          String text :=
39                            [$doctorsTa.row$.doctorProfessional.name
40                              .concat(' ')
41                              .concat($doctorsTa.row$.doctorProfessional.surname)] } } }

42 Button movePatientWi.moveBu {
43   event onClick {
44         if [not $departmentsTa.selected$.oclIsUndefined()] {
45           if[not $doctorsTa.selected$.oclIsUndefined()] {
46            [$movePatientWi.patient$.department] := [$departmentsTa.selected$]
47            [$movePatientWi.patient$.doctor] := [$doctorsTa.selected$]
48            notification(['Success'],['The patient has been reassigned.'],[0]) }
49           else {
50            notification(['Error'],['Please, select first a doctor.'],[0]) } }
51         else {
52           notification(['Error'],['Please, select first a department.'],[0]) } } } }

53 Button movePatientWi.backBu {
54   event onClick {
55         back } }
```

Fig. 5. A window for reassigning a selected patient (part II)

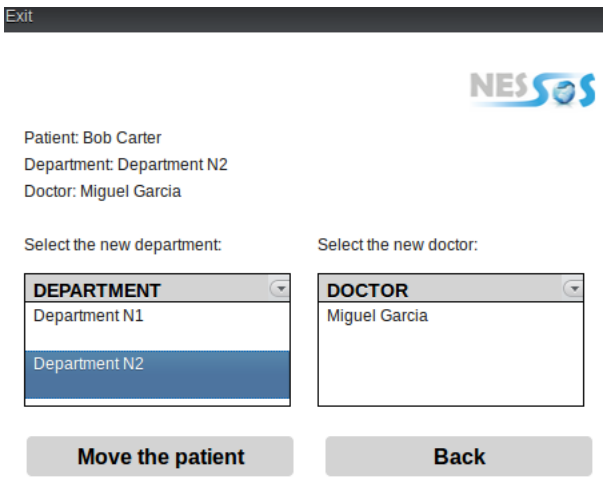


Fig. 6. Screenshot of the window for reassigning a selected patient

```
46.1  if [[$movePatientWi.role$ = ADMINISTRATOR
46.2        and $movePatientWi.caller$.worksIn
46.3              →includes($departmentsTa.selected$.belongsTo)] {
46.4      [$movePatientWi.patient$.department] := [$departmentsTa.selected$] }
46.5      else { fail }

47.1  if [[$movePatientWi.role$ = ADMINISTRATOR
47.2        and $movePatientWi.caller$.worksIn→exists(m|
47.3              $doctorsTa.selected$.doctorProfessional.worksIn→includes(m))
47.4            and $movePatientWi.caller$.worksIn
47.5              →includes($movePatientWi.patient$.department.belongsTo))]  {
47.6      [$movePatientWi.patient$.doctor] := [$doctorsTa.selected$] }
47.7      else { fail }
```

Fig. 7. The security-aware actions for reassigning a selected patient

specify the system's intended behavior, one should not expect the generated system to do so either. *Quod natura non dat, Salmantica non praestat.*[5] Experience shows that even when using powerful, high-level modeling languages, it is easy to make logical errors and omissions. It is critical not only that the modeling language has a well-defined semantic, so one can know what one is doing, but also that there is tool support for analyzing the modeled systems' properties.

In this section we explain how we can reason about an important property of ActionGUI models, called *data invariant preservation*. We use the EHRM application for illustration.

4.1 Data invariant preservation

We first introduce some terminology. Recall that the actions triggered by an event may be specified using if-then-else statements. At execution time, the exact sequence of actions taken is determined by how the different conditions of each if-then-else statements are evaluated in the system's state at the time of evaluation. Note that this state includes both the state of the persistence layer and the state of the GUI, in particular, its widget variables. Since each action may update the system's state, a sequence of actions gives rise to a sequence of states, which we call an *execution path*.

ActionGUI's data model may include *data invariants*. We have given several examples of these in Section 3.2. These are properties that are *required* to be satisfied in every (reachable) system state. Invariance of a property must be *proven* and the standard way to do this is to show that the property is inductive, that is, it is satisfied in the system's initial state and, whenever it is satisfied in a state, it is satisfied in all possible successor states. Below we shall focus on the inductive step: proving invariant preservation.

[5] Less elegantly said, *garbage in, garbage out*.

Formally, let Φ be a collection of data invariants. An event *preserves a data invariant* $\phi \in \Phi$ if and only if for every execution path triggered by the event, if every data invariant $\psi \in \Phi$ is satisfied at the initial state of the execution path, then ϕ is also satisfied at the final state. Here we leverage ActionGUI's transaction semantics and that transactions are implemented in a way that ensures their atomicity: The intermediate states of an execution path may be considered to be internal and may therefore (temporarily) violate ψ. An event is *Φ-data invariant preserving* when it preserves all data invariants in Φ.

Our proof procedure, illustrated below, is based on the fact that each event defines an *action tree*. The nodes in this tree are the actions triggered by the event and branching corresponds to the if-then-else conditions governing the execution of these actions. As expected, every successful transaction corresponds to executing a sequence of actions given by one of the branches of the action tree, from the root to a leaf. Note that, to simplify our exposition we omit both iteration statements and event-triggering actions; including these would lead to action graphs rather than trees.

1	*Each patient is treated by a doctor.* Patient.allInstances()→forAll(p|not(p.doctor.oclIsUndefined()))
2	*Each patient is treated in a department.* Patient.allInstances()→forAll(p|not(p.department.oclIsUndefined()))
3	*Each patient is treated by a doctor who works for a set of departments that includes the department where the patient is treated.* Patient.allInstances()→forAll(p| p.doctor.doctorDepartments→includes(p.department))

Fig. 8. Examples of the EHRM data model's invariants.

Reassigning doctors and departments to patients We show in Figure 9 the action tree defined by the the button moveBu's event **onClick**. For ease of later reference, we assign labels for the actions and the if-then-else conditions. Note that:

- Branch 1 corresponds to the case when a department and a doctor are both selected when the button moveBU is clicked-on. In this situation, the patient will be first assigned to the selected department, and then to the selected doctor; finally, a message confirming these actions will be displayed.
- Branch 2 corresponds to the case when a department is not selected when the button moveBU is clicked-on. In this situation, a message stating that a department must be first selected will be displayed.
- Branch 3 corresponds to the case when a department is selected, but a doctor is not, when the button moveBU is clicked-on. In this situation, a message stating that a doctor must first be selected will be displayed.

Next, we use this action tree to reason about whether the button moveBu's event **onClick** preserves the data invariants 1–3 listed in Figure 8.

Actions

assign_dept	= [$movePatientWi.patient$.department] := [$departmentsTa.selected$]
assign_doctor	= [$movePatientWi.patient$.doctor] := [$doctorsTa.selected$]
notify_reassign	= notification(['Success'],['The patient is reassigned.'],[0])
error_select_doctor	= notification(['Error'],['Select first a doctor.'],[0])
error_select_dept	= notification(['Error'],['Select first a department.'],[0])

If-then-else conditions

a_dept_is_selected	= not $departmentsTa.selected$.oclIsUndefined()
a_doctor_is_selected	= not $doctorsTA.selected$.oclIsUndefined()

Branch 1

a_dept_is_selected = true ∧ *a_doctor_is_selected* = true	
nodes	actions
1	*assign_dept*
2	*assign_doctor*
3	*notify_reassignment*

Branch 2

a_dept_is_selected = false	
nodes	actions
1	*error_select_dept*

Branch 3

a_dept_is_selected = true ∧ *a_doctor_is_selected* = false	
nodes	actions
1	*error_select_doctor*

Fig. 9. Action tree for the button moveBU's **onClick**.

Branch 1: Data invariants 1 and 2. Recall that these data invariants state that every patient is assigned to exactly one doctor and one department. Observe that the initial state in every successful transaction in this branch will satisfy the conditions *a_dept_is_selected* and *a_doctor_is_selected*. Therefore the arguments of the actions *assign_dept* and *assign_doctor* will necessarily not be null (neither invalid) when these actions are called. Thus, the conditions *a_dept_is_selected* and *a_doctor_is_selected*, together with the postconditions of the actions *assign_dept* and *assign_doctor*, guarantee that every successful transaction in this branch preserves the data invariants 1 and 2.

Branch 1: Data invariant 3. Recall that this data invariant states that every patient is assigned to a department where its doctor works. Interestingly, there

is no guarantee that every successful transaction in this branch preserves the data invariant 3. This is because the doctors shown in the table doctorsTa are those belonging to the selected department at the time of this selection (line 19–21 in Figure 4); however, there is no guarantee that, by the time the user clicks on the button moveBu, this relationship still holds for the selected doctor.

To guarantee that data invariant 3 is preserved by every successful transactions in this branch, we can simply enclose the sequence of actions *assing_dept*, *assig_dept*, and *notifiy_reassignment* (lines 46-54 in Figure 5) within an (additional) if-then-else with the following condition:

$departmentsTa.selected$.doctors→includes($doctorsTa.selected$).

Branch 2 and 3. Since these branches do not contain any data actions, every successful transaction in these branches will trivially preserve all the data model's invariants.

We conclude this section by summarizing in Figure 10(a) our analysis of data invariant preservation for the button moveBu's event **onClick**. For the sake of illustration, we also consider in Figures 10(b) and 10(c) data invariant preservation for two modified versions of the button moveBu's event **onClick**. In the first case, we have removed the innermost if-then-else, i.e., the one whose condition checks that a doctor has been selected. In the second case, we have removed the outermost if-then-else, i.e., the one whose condition checks that a department has been selected. As expected, if we remove the innermost if-then-else, there is no guarantee that data invariant 1, i.e., that every patient is assigned to exactly one doctor, will be preserved. Similarly, if we remove the outermost if-then-else, there is no guarantee that data invariant 2, i.e., that every patient is assigned to exactly one department, will be preserved.

	Branches		
Invs.	1	2	3
1	✓	✓	✓
2	✓	✓	✓
3	✗	✓	✓

(a) Original

	Branches	
Invs.	1	2
1	✗	✓
2	✓	✓
3	✗	✓

(b) Without *a_dept_is_selected*

	Branches	
Invs.	1	2
1	✓	✓
2	✗	✓
3	✗	✓

(c) Without *a_doctor_is_selected*

Fig. 10. Checking data invariants preservation for different versions of the button moveBu's **onClick**.

4.2 Checking data invariant preservation

We now describe how we check whether modeled events preserve data invariants.

Fix a data model D and a GUI model G. Let Φ be D's declared invariants. Let ev be an event in G and let B be a branch of ev's action tree containing n actions. To check that every instance of B preserves the invariants in Φ, we proceed as follows:

1. We define a ComponentUML data model D_n that represents all sequences of n states. Recall that a state is any instance of the data model D along with any assignment to the widget variables in G.
2. For $1 \leq i < n$, we formalize an OCL expression, in the context of D_n, that the i-th action's postconditions are satisfied in the $(i+1)$-th state. We denote by $\mathit{Posts}(B)$ the resulting set of OCL expressions.
3. For $1 \leq i \leq n$, we formalize an OCL expression, in the context of D_n, that the *guard* of the i-th action is satisfied in the i-th state. We denote by $\mathit{Guards}(B)$ the resulting set of OCL expressions.
4. For each invariant $\phi \in \Phi$, we formalize an OCL expression, in the context of D_n, that ϕ is satisfied in the first state (initial state). We denote by $\Phi(1)$ the resulting set of OCL expressions.
5. For each invariant $\phi \in \Phi$, we formalize an OCL expression $\psi(n)$, in the context of D_n, stating that ψ is satisfied in the n-th (final) state.
6. We prove that there is no instance of D_n that satisfies

$$\Phi(1) \cup \mathit{Posts}(B) \cup \mathit{Guards}(B) \ \cup \{\neg\psi(n)\}\,.$$

This formula expresses that there is no sequence of n states where the first state satisfies all the invariants, each state satisfies the postcondition of the action leading to it, each state satisfies the condition that guards the action leading to the next state, and the final state does not satisfy ψ.

We have built a tool that implements the above steps. For every data model D with invariants Φ, GUI model G, and event ev in G, our tool automatically generates the set of branches Π corresponding to ev. Then, for each branch $B \in \Pi$ and invariant $\psi \in \Phi$, it generates the data model D_n and the sets of OCL expressions $\Phi(1)$, $\mathit{Posts}(B)$, $\mathit{Guards}(B)$, and $\{\neg\psi(n)\}$, where n is B's length. Finally, our tool uses the mapping OCL2FOL$^+$ [8] to generate the first-order proof-score corresponding to step 6 above, both in SMT-LIB syntax [2] and DFG syntax [14].

4.3 Analyzing the EHRM application

We report here on preliminary experiments where we used our tool to check data invariant preservation for the EHRM application. The application's full GUI model only contains 8 events whose associated statements include data actions, and therefore must be checked. Moreover, the action trees defined by these events contain 49 branches in total, but only 8 of these branches include data actions. Therefore, since the full EHRM application's data model contains 66 invariants, we must perform a total of 528 checks (8 branches $\times$ 66 invariants) to prove data invariant preservation for this application.

We ran these checks on a laptop computer, with a 2.66GHz Intel Core 2 Duo processor and 4Gb 1067MHz. memory, using SPASS [15] as the back-end theorem-prover. Here we summarize the results. First, for branches containing up to 3 data actions (50% of the non-trivial checks fall into this category, including our running example) checking takes less than 10 milliseconds to return "proof found" when the invariants are preserved. Second, when checking branches containing 8-10 actions and 8-10 conditions (45% of the non-trivial checks), we also obtain "proof found" in less than 30 seconds when the invariants are preserved, except for some complex invariants where checking takes up to 3 minutes. Third, for a branch containing 30 actions and 6 conditions, cheking also takes less than 40 seconds to return "proof found" when the invariants are preserved, except again for some complex invariants where it takes up to 5 minutes.

Finally, note that all these results depend on the interaction between (i) the way we formalize sequences of n states, OCL invariants, actions' guards, and actions' post-conditions, and (ii) the heuristics implemented in the verification back-end we use, here SPASS. We are currently analyzing this interaction in depth to better understand the scope and limitations of our tool. For example, we already know that SPASS seems not able to return "completion found" (we timed out after four days) when, for the sake of experiment, we remove some conditions from the branches, thereby violating some of the invariants.

5 Conclusions

This chapter complements the article [3], where we present the ActionGUI methodology and tool in detail. [3] also contains an extensive comparison with related work and provides summary statistics from five different developments. The eHealth application was one of the smallest examples considered there and other examples are roughly an order of magnitude larger, e.g., with hundreds of windows, buttons, labels, and if-then-else statements and thousands of OCL statements. In contrast, in this paper, we present one case study in detail. We also describe model-based property checking, which was not addressed in [3].

Among the methodologies and tools reviewed in [3], UWE [7, 6, 11] and ZOOM [10] are the most closely related to our work. As a modeling tool, UWE provides the modeler with a higher-level of abstraction than ActionGUI. In particular, the actions executed by the widgets' events are described in UWE using natural language. Thus, unless the models are appropriately refined, as discussed in [11], UWE does not support code-generation. In contrast, UWE provides specific diagrams for modeling GUI *presentations* and *navigations*, which facilitate the task of GUI modeling. [6] extends UWE to use SecureUML for modeling security policies. However, this work does not use model-transformation to lift automatically the security policy to the GUI level. Instead the UWE modeler is responsible for adding all the appropriate authorization checks to the GUI model. Like ActionGUI, ZOOM allows GUI modelers to specify widgets, their events, and their actions. Moreover, using an extension of Z [16], one can specify the conditions of the actions and their arguments, similar to how this is done

in ActionGUI using OCL. In contrast to ActionGUI, ZOOM does not provide a language for modeling security and security aspects are not explicitly considered in this approach. Moreover, ZOOM does not support code-generation. It only provides interpreters for model animation.

In the following we draw some conclusions based on our experience with the eHealth application and developing other applications with ActionGUI. First, ActionGUI's security modeling language is well suited for modeling access control policies that combine both *declarative* and *programmatic* aspects. Declarative access control policies depend on static information, namely the assignments of users and permissions to roles. Programmatic access control depends on dynamic information, namely the satisfaction of authorization constraints in the current system state. Programmatic access control is formalized using authorization constraints and, as Section 3.3 illustrates, this allows us to model directly the kinds of authorization rules considered in the eHealth case study.

Second, ActionGUI's graphical user interface modeling language is well suited for modeling *dynamic web pages*. These are pages, displayed at the client, that are generated at the time of access by a user or that change as a result of user interaction. As Section 3.4 illustrates, an important aspect of our methodology is that developers can model this behavior independent of the access control policy. The policy is later lifted from the security model to this behavioral model, as described in Section 3.5.

Third, as explained in Section 3.6, the ActionGUI code generator can automatically generate ready-to-deploy, security-aware, data-management web applications. By data-management, we mean that most of the behavior described in the GUI model is built from CRUD actions (which create, read, update and delete data). When all behavior can be described this way, then the entire application can be generated from the models, including a complete, configured security infrastructure and back-end database support.

Finally, our case study illustrates how users can specify properties of ActionGUI models, such as invariant preservation. Moreover, as described in Section 4, our approach to checking these properties based on translation to first-order logic is practical, see also [5]. This is a form of model-checking and, as in other domains, it has an important role to play in building and certifying security-critical systems. Designers and system certifiers can reason about systems at the model level using automated tool support. Moreover, with our approach, they can afterwards generate model-conform, and therefore property-conform, systems simply by pressing a button. Our experience with ActionGUI shows that this is not merely a vision for the future, but it is realizable today, at least for small and medium-scale data-management applications.

Acknowledgements

This work is partially supported by the EU FP7-ICT Project "NESSoS: Network of Excellence on Engineering Secure Future Internet Software Services and Sys-

tems" (256980) and by the Spanish Ministry of Economy and Competitiveness Project "StrongSoft" (TIN2012-39391-C04-04).

References

1. ActionGUI. The ActionGUI project, 2013. `http://www.actiongui.org`.
2. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.
3. D. Basin, M. Clavel, M. Egea, M. A. G. de Dios, and C. Dania. A model-driven methodology for developing secure data-management applications. *IEEE Transactions on Software Engineering*, 2014. To appear.
4. D. Basin, J. Doser, and T. Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology*, 15(1):39–91, 2006.
5. D. A. Basin, M. Clavel, and M. Egea. A decade of model-driven security. In *Proceedings of the 16th ACM symposium on Access control models and technologies (SACMAT 2011)*, volume 1998443, pages 1–10, Innsbruck, Austria, 2011. New York, NY, USA.
6. M. Busch. Integration of security aspects in web engineering. Master's thesis, Institut für Informatik, Ludwig-Maximilians-Universität, München, Germany, 2011.
7. M. Busch and N. Koch. MagicUWE - a case tool plugin for modeling web applications. In M. Gaedke, M. Grossniklaus, and O. Díaz, editors, *Proc. of ICWE'09*, volume 5648 of *LNCS*, pages 505–508. Springer, 2009.
8. C. Dania and M. Clavel. OCL2FOL+: Coping with Undefinedness. In J. Cabot, M. Gogolla, I. Ráth, and E. D. Willink, editors, *OCL@MoDELS*, volume 1092 of *CEUR Workshop Proceedings*, pages 53–62. CEUR-WS.org, 2013.
9. D. F. Ferraiolo, R. S. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.
10. X. Jia, A. Steele, L. Qin, H. Liu, and C. Jones. Executable visual software modeling—the ZOOM approach. *Software Quality Control*, 15:27–51, March 2007.
11. C. Kroiss, N. Koch, and A. Knapp. UWE4JSF: A model-driven generation approach for web applications. In M. Gaedke, M. Grossniklaus, and O. Díaz, editors, *Proc. of ICWE'09*, volume 5648 of *LNCS*, pages 493–496. Springer, 2009.
12. NESSoS. The European Network of Excellence on Engineering Secure Future internet Software Services and Systems, 2010. `http://www.nessos-project.eu`.
13. Object Management Group. Object constraint language specification version 2.3.1. Technical report, OMG, 2012. `http://www.omg.org/spec/OCL/2.3.1`.
14. C. Weidenbach. SPASS input syntax version 1.5, 1999.
15. C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. In R. A. Schmidt, editor, *CADE*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer, 2009.
16. J. Woodcock and J. Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

ActionGUI semantics

David Basin[1], Manuel Clavel[2,3], Marina Egea[4], Miguel A. García de Dios[2], and Carolina Dania[2,3]

[1] ETH Zürich, Switzerland
basin@inf.ethz.ch
[2] IMDEA Software Institute, Madrid, Spain
[manuel.clavel,miguelangel.garcia,carolina.dania]@imdea.org
[3] Universidad Complutense, Madrid, Spain
[4] Atos Research & Innovation Dept., Madrid, Spain
marina.egea@atosresearch.eu

Abstract. In this technical report we provide the formal account of our ActionGUI methodology, including the semantics of the modeling languages that we use, the definition of our many-models-to-models transformation, and the proof of its correctness.

1 ComponentUML

In this section we first define ComponentUML *data models*. Then, given a ComponentUML data model D, we define *D-object models*. Finally, we define the *semantics* of a ComponentUML data model D as the set $\mathrm{Sem}(D)$ of all the D-object models.

Notation. Let $\mathrm{TP} = \{\mathsf{Integer}, \mathsf{Real}, \mathsf{String}, \mathsf{Boolean}\}$ be the set of ComponentUML primitive data types. In what follows, we denote by $[\![t]\!]^{\mathrm{TP}}$ the standard carrier set of t, for each primitive data type $t \in \mathrm{TP}$, e.g., $[\![\mathsf{Integer}]\!]^{\mathrm{TP}} = \mathbb{Z}$.

Let $\mathcal{A} \subset [\![\mathsf{String}]\!]^{\mathrm{TP}}$ be the set of all finite strings that only contain letters of the English alphabet.

1.1 ComponentUML data models

Definition. A ComponentUML *data model* is a tuple $\langle C, AT, AS, ASO\rangle$ such that:

- $C \subset \mathcal{A}$ is a set of class identifiers.
- AT is a set of triples $\langle at, c, t\rangle$, also represented as $at_{(c,t)}$, where $at \in \mathcal{A}$ is an attribute identifier, $c \in C$, $t \in C \cup \mathrm{TP}$, and c and t are, respectively, the class and the type of the attribute at.
- AS is a set of triples $\langle as, c, c'\rangle$, also denoted as $as_{(c,c')}$, where $as \in \mathcal{A}$ is an association-end identifier, $c, c' \in C$, and c and c' are, respectively, the source and the target classes of as.
- ASO is a symmetric relation, $ASO \subseteq AS \times AS$, where $(as_{(c,c')}, as'_{(c',c)}) \in ASO$ represents that as' is the association-end opposite to as, and vice versa, and $c, c' \in C$.

Invariants.

- There is no class whose identifier also belongs to TP.
- Attributes and associations-ends of the same class always have different identifiers.
- Every association-end is related with exactly another association-end. That is, for every tuple $\langle as, c, c' \rangle$ in AS, there exists exactly one other tuple $\langle as', c', c \rangle$ in AS such that $(\langle as, c, c' \rangle, \langle as', c', c \rangle)$ in ASO.

1.2 ComponentUML object models

Definition. Let D be a ComponentUML data model $\langle C, AT, AS, ASO \rangle$. Then, a *$D$-object model* is a tuple $\langle O, VA, LK \rangle$, such that:

- O is a set of pairs $\langle o, c \rangle$, where $o \in \mathcal{A}$ is an object identifier and $c \in C$. Each pair $\langle o, c \rangle$, also represented as o_c, denotes that the object o is of the class c.
- VA is a set of triples $\langle o_c, at_{(c,t)}, va \rangle$, where $at_{(c,t)} \in AT$, $o_c \in O$, $t \in \mathrm{TP}$, and $va \in [\![t]\!]^{\mathrm{TP}}$ is a value of type t. Each triple $\langle o_c, at_{(c,t)}, va \rangle$ denotes that va is the value of the attribute at of the object o.
- LK is a set of triples $\langle o_c, as_{(c,c')}, o'_{c'} \rangle$, where $as_{(c,c')} \in AS$, and $o_c, o'_{c'} \in O$. Each tuple $\langle o_c, as_{(c,c')}, o'_{c'} \rangle$ denotes that the object o' is among the objects that are linked to the object o through the association-end as.

Invariants.

- There are no two different values for the same attribute of the same object. (However, it is not necessary that every attribute of an object has a value.)
- For every association-end $as_{(c,c')}$ in AS, such that $(as_{(c,c')}, as'_{(c',c)})$ in ASO, if there is a link $\langle o_c, as_{(c,c')}, o'_{c'} \rangle \in LK$ between two objects o_c and $o'_{c'}$ through this association-end, then there is also a link $\langle o'_{c'}, as'_{(c',c)}, o_c \rangle \in LK$ between these two objects through the opposite association-end.

1.3 Semantics of ComponentUML data models

Definition. Let D be a ComponentUML data model. The *semantics* of D, denoted by $\mathrm{Sem}(D)$, is the set of all the ComponentUML D-object models.

2 SecureUML

In this section we first define SecureUML *data actions* and SecureUML *authorization constraints*, both relative to a given ComponentUML data model. Then, we define SecureUML *security models*, also relative to a given ComponentUML model. Next, given a SecureUML security model S, we define *S-authorized actions*. Finally, we define the *semantics* of a SecureUML security model as the set $\mathrm{Sem}(S)$ of all the S-authorized actions and, based on this definition, we define the notion of a *consistent* SecureUML security model.

Notation. Let D be a data model, and let I be a D-object model, $I = \langle O, VA, LK \rangle$. In what follows, we will use the following notation:

- We denote by $\mathrm{Typ}(D)$ the set of all the OCL types, given the classes declared in D. These types are defined in the OCL standard [1].
- We denote by $\mathrm{Expr}(D)$ the set of all the OCL expressions that have D as their contextual model. These expressions are defined in the OCL standard [1]. Note that, by definition, they do not contain free variables.
- Let X be a set of pairs $\langle x, t \rangle$, also written as x_t, where $x \in \mathcal{A}$ is a variable identifier of type $t \in \mathrm{Typ}(D)$. Then, we denote by $\mathrm{Expr}(D, X)$ the set of all the OCL expressions that have D as their contextual model but that may also contain variables in X as free variables. Moreover, for every $expr \in \mathrm{Expr}(D, X)$, we denote by $\mathrm{FVar}(expr) \subseteq X$ the set of all the free variables contained in $expr$.
- We denote by $\mathrm{Expr}(D_I)$ the set of all the OCL expressions that have D as their contextual model and may also contain as constants the objects $o_c \in O$.
- Let $expr$ be an OCL expression in $\mathrm{Expr}(D_I)$. Then, we denote by $[\![expr]\!]^I$ the evaluation of the expression $expr$ in the object model I, as defined in the OCL standard [1]. Note that the evaluation of an OCL expression always return a *literal expression* in $Expr(D_I)$, which can not be further reduced and which does not contain any variables.
- Let X be a set of variables x_t, where $t \in \mathrm{Typ}(D)$. Then, a (X, I)-*substitution* θ is a function, $\theta : X \to \mathrm{Expr}(D_I)$, that assigns to each variable in X an expression in $\mathrm{Expr}(D_I)$ of the appropriate type.
  Now, let $x_t \in X$ be a variable and let $expr$ in $\mathrm{Expr}(D_I)$ be an expression of type t. Then, we denote by $\theta \oplus \{x_t \mapsto expr\}$, $\theta \oplus \{x_t \mapsto expr\} : X \to \mathrm{Expr}(D_I)$, the *overriding* of θ by $\{x_t \mapsto expr\}$. That is, $\theta \oplus \{x_t \mapsto expr\}(x_t) = expr$, but, for every other $x'_{t'} \in X$, $x_t \neq x'_{t'}$, $\theta \oplus \{x_t \mapsto expr\}(x'_{t'}) = \theta(x'_{t'})$.
  Moreover, for every (X, I)-*substitution* θ, we denote as $\hat{\theta}$ the homomorphic extension of θ over the set $\mathrm{Expr}(D, X)$. Finally, for $expr \in \mathrm{Expr}(D, X)$, we write $\hat{\theta}(expr)$ as $(expr)\theta$.

2.1 SecureUML data actions

Definition. Let D be a data model $\langle C, AT, AS, ASO \rangle$. Then, we denote by $\mathrm{Act}(D)$ the set of all the (atomic) *data actions* that can be executed on D-object models. $\mathrm{Act}(D)$ is defined as follows: for every class $c \in C$, every attribute $at_{(c,t)} \in AT$, and every association-end $as_{(c,c')} \in AS$,

$$\mathsf{Create}(c), \mathsf{Delete}(c) \in \mathrm{Act}(D).$$
$$\mathsf{Read}(at_{(c,t)}), \mathsf{Update}(at_{(c,t)}) \in \mathrm{Act}(D).$$
$$\mathsf{Read}(as_{(c,c')}), \mathsf{Create}(as_{(c,c')}), \mathsf{Delete}(as_{(c,c')}) \in \mathrm{Act}(D).$$

The notation $action(resource)$ reflects that data actions are always upon resources.

2.2 SecureUML authorization constraints

Definition. Let D be a data model $\langle C, AT, AS, ASO\rangle$. Let $u \in C$ be a class that represents the *users*. Let $act \in Act(D)$ be a data action. Then, we denote by $\mathrm{AuthExpr}(D, u, act)$ the set of all the *authorization constraints* that can be imposed on users of type u for executing the data action act with respect to the data model D. Informally, an authorization constraint is an (extended) OCL expression that may contain distinguished keywords (logically interpreted as free variables) that refer to the user attempting to execute the action (caller), to the data upon which the action is to be executed (self), or to the data that the action takes as its arguments (value and target).

More formally, $\mathrm{AuthExpr}(D, u, act)$ is defined as follows: for every class $c \in C$, every attribute $at_{(c,t)} \in AT$, and every association-end $as_{(c,c')} \in AS$:

$$\mathrm{AuthExpr}(D, u, \mathsf{Create}(c)) = \mathrm{Expr}(D, \{\mathsf{caller}_u\}).$$
$$\mathrm{AuthExpr}(D, u, \mathsf{Delete}(c)) = \mathrm{Expr}(D, \{\mathsf{self}_c, \mathsf{caller}_u\}).$$
$$\mathrm{AuthExpr}(D, u, \mathsf{Read}(at_{(c,t)})) = \mathrm{Expr}(D, \{\mathsf{self}_c, \mathsf{caller}_u\}).$$
$$\mathrm{AuthExpr}(D, u, \mathsf{Update}(at_{(c,t)})) = \mathrm{Expr}(D, \{\mathsf{self}_c, \mathsf{value}_t, \mathsf{caller}_u\}).$$
$$\mathrm{AuthExpr}(D, u, \mathsf{Read}(as_{(c,c')})) = \mathrm{Expr}(D, \{\mathsf{self}_c, \mathsf{caller}_u\}).$$
$$\mathrm{AuthExpr}(D, u, \mathsf{Create}(as_{(c,c')})) = \mathrm{Expr}(D, \{\mathsf{self}_c, \mathsf{target}_{c'}, \mathsf{caller}_u\}).$$
$$\mathrm{AuthExpr}(D, u, \mathsf{Delete}(as_{(c,c')})) = \mathrm{Expr}(D, \{\mathsf{self}_c, \mathsf{target}_{c'}, \mathsf{caller}_u\}).$$

2.3 SecureUML security models

Definition. Let D be a data model $\langle C, AT, AS, ASO\rangle$. Then, a SecureUML *$D$-security model* S is a tuple $\langle D, R, RH, u, P\rangle$ such that:

- $R \subset \mathcal{A}$ is a set of role identifiers.
- $RH \subset R \times R$ is a partial order representing the role hierarchy.
- $u \in C$ is a class that represents the *users*.
- P is a set of triples $\langle r, act, expr\rangle$ representing *permissions*: namely, that the role $r \in R$ is granted permission for the action $act \in \mathrm{Act}(D)$ provided the constraint $expr \in \mathrm{AuthExpr}(D, u, act)$ is satisfied.

Definition. Let D be a data model $\langle C, AT, AS, ASO\rangle$. Let S be a D-security model $\langle D, R, RH, u, P\rangle$. Then, $\mathrm{AuthPerm}(S, r, act)$ is the disjunction of all the authorization constraints controlling the access for users in the role r to execute the action act, according to S. $\mathrm{AuthPerm}(S, r, act)$ is defined as follows: Let $Q = \{expr \mid \exists r' \in R.\, \langle r', act, expr\rangle \in P \wedge (r, r') \in RH\}$. Then,

$$\mathrm{AuthPerm}(S, r, act) = \begin{cases} expr_1 \text{ or } \ldots \text{ or } expr_n, & \text{if } Q = \{expr_1, \ldots, expr_n\}. \\ \mathsf{false}, & \text{if } Q = \emptyset. \end{cases}$$

Note that, by definition, $\mathrm{AuthPerm}(S, r, act) \in \mathrm{Expr}(D, X)$, where X is the set containing caller_u plus the appropriate instances of self, target, and value, depending on the type of the action act.

2.4 SecureUML authorized actions

Definition. Let D be a data model $\langle C, AT, AS, ASO\rangle$. Let S be a D-security model $\langle D, R, RH, u, P\rangle$. Let I be a D-object model $\langle O, VA, LK\rangle$. Let $o_u \in O$ be a user, $r \in R$ be a role, and $act \in \mathrm{Act}(D)$ be a D-data action. Moreover, let θ be a $(\mathrm{FVar}(\mathrm{AuthPerm}(S, r, act)), I)$-substitution.

Then, $\langle I, o_u, r, act, \theta\rangle$ is an *S-authorized action* if and only if

$$[\![\mathrm{AuthPerm}(S, r, act)(\theta \oplus \{\mathsf{caller}_u \mapsto o_u\})]\!]^I = \mathsf{true}.$$

Note that, given our definition of AuthPerm,

- No permission is granted for executing an action, unless it is explicitly declared.
- All permissions are inherited along the role hierarchy.

2.5 Semantics of SecureUML security models

Definition. Let D be a data model and let S be a D-security model. Then, the *semantics* of S, given by $\mathrm{Sem}(S)$, is the set of all the S-authorized actions.

3 GUIML

In this section we first define GUIML *layout models*, which simply model graphical user interfaces without considering their behaviors. Then, we define GUIML *statements*, which specify sequences of actions that are possibly conditional and iterated. Next, we define GUIML *behavioral models*, which are GUIML layout models but also with associated behavior, i.e., with statements associated to each of the widget events. Finally, we define a set of inference rules that will provide the (operational) *semantics* of GUIML *events* as the set $\mathrm{Sem}(G, ev)$ of all the *transitions* defined by these rules.

Notation. In what follows, let ET be the set of GUIML event types,

$$\mathrm{ET} = \{\mathsf{onClick}, \mathsf{onCreate}\}.$$

Also, let WT be the set of GUIML widget types,

$$\mathrm{WT} = \{\mathsf{Window}, \mathsf{Table}, \mathsf{Combo\text{-}box}, \mathsf{Button}, \mathsf{Text\ field}, \mathsf{Label}, \mathsf{Boolean\ check}\}.$$

3.1 GUIML layout models

Definition. A GUIML *layout model* H is a tuple $\langle W, WC, X, EV\rangle$ such that:

- W is a set of pairs $\langle w, wt\rangle$, also represented as w_{wt}, where $w \in \mathcal{A}$ is a widget identifier, and $wt \in \mathrm{WT}$ is the widget's type.

- $WC \subset W \times W$ is a relation representing the widget containment.
- X is a set of pairs $\langle\langle x, t\rangle, \langle w, wt\rangle\rangle$, called *widget variables*, also represented as $\langle x_t, w_{wt}\rangle$, where x is a variable identifier, $t \in \mathrm{Typ}(D)$ is the variable's type, and $\langle w, wt\rangle \in W$ is the widget that *owns* this variable.
- EV is a set of pairs $\langle ev, \langle w, wt\rangle\rangle$, also represented as $\langle ev, w_{wt}\rangle$, where $ev \in \mathrm{ET}$ is an event type and $\langle w, wt\rangle \in W$ is the widget that supports this event type.

Invariants.

- The containment relation WC defines set of rooted trees. Moreover, at the root of every tree in WC there is a widget of type Window and, conversely, every widget in W of type Window is the root of a tree in WC.
- There are no two variables owned by the same widget with the same identifier.
- If two widgets are directly contained in the same widget, then they have different identifiers.

Notation. Let H be a GUIML layout model $\langle W, WC, X, EV\rangle$. In what follows we will use the following notation:

- We denote by WC^+ the transitive closure of the containment relation defined in WC.
- Let $w_{wt} \in W$ be a widget in W, $wt \neq \mathsf{Window}$. Then, we denote by $\mathrm{Win}(H, w_{wt})$ the window that contains w_{wt} in W, i.e., $(w_{wt}, \mathrm{Win}(H, w_{wt})) \in WC^+$.
- Let $w_{wt} \in W$ be a widget in W. Then, we denote by $\mathrm{Var}(H, w_{wt})$ the set of variables in X that are owned by w_{wt}, i.e., $\mathrm{Var}(H, w_{wt}) = \{\langle x_t, w'_{wt'}\rangle \mid \langle x_t, w'_{wt'}\rangle \in X \wedge w_{wt} = w'_{wt'}\}$.
- Let $w_{wt} \in W$ be a widget in W. Then, we denote by $\mathrm{Var}^\sharp(H, w_{wt})$ the set of the variables in X that are *visible* from w_{wt}. $\mathrm{Var}^\sharp(H, w_{wt})$ is defined as follows:

$$\begin{aligned}\mathrm{Var}^\sharp(H, w_{wt}) = \mathrm{Var}(H, w_{wt}) \cup \\ \{\langle x_t, w'_{wt'}\rangle \mid \langle x_t, w'_{wt'}\rangle \in \mathrm{Var}(H, w'_{wt'}) \wedge \\ (w'_{wt'}, \mathrm{Win}(H, w_{wt})) \in WC^+\}.\end{aligned}$$

  Note that, by definition, if two widgets are contained in the same window, then their sets of visible variables are identical. Also, the set of visible variables of a widget is the same than the set of visible variables of the widget's containing window.

3.2 GUIML statements

Definition. Let D be a data model $\langle C, AT, AS, ASO\rangle$. Let H be a GUIML layout model $\langle W, WC, X, EV\rangle$. Let w_{wt} be a window in H, i.e., $w_{\mathsf{Window}} \in W$. Then, we denote by $\mathrm{Stm}(D, H, w)$ the set of all the *statements* that can be written in the context of the window w. This set is inductively defined as follows:

Base case (data actions): The building block for statements are the data actions along with the GUI actions. The GUIML data actions are the SecureUML data actions introduced before, except that they now take additional arguments that, depending on the action's type, either specify, using OCL (extended with widget variables), the object *self* upon which the action is to be executed, or the *value* and *target* of this action, or indicate the widget *variable* where the action's outcome is to be stored. To reflect this difference between the GUIML data actions and their corresponding SecureUML data actions, we use the notation $action(resource)[\boldsymbol{arguments}]$ for GUIML data actions. Thus, if $action(resource)[\boldsymbol{arguments}]$ is a GUIML data action, then $action(resource)$ is its corresponding SecureML data action.

- For every entity create action $\mathsf{Create}(c) \in \mathrm{Act}(D)$ and every *variable* of type c in $\mathrm{Var}^\sharp(H, w_{wt})$, then
$$\mathsf{Create}(c)[variable] \in \mathrm{Stm}(D, H, w).$$
- For every entity delete action $\mathsf{Delete}(c) \in \mathrm{Act}(D)$ and every expression *self* of type c in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, then
$$\mathsf{Delete}(c)[self] \in \mathrm{Stm}(D, H, w).$$
- For every attribute read action $\mathsf{Read}(at_{(c,t)}) \in \mathrm{Act}(D)$, every expression *self* of type c in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, and every widget *variable* of type t in $\mathrm{Var}^\sharp(H, w_{wt})$, then
$$\mathsf{Read}(at_{(c,t)})[self, variable] \in \mathrm{Stm}(D, H, w).$$
- For every attribute update action $\mathsf{Update}(at_{(c,t)}) \in \mathrm{Act}(D)$, every expression *self* of type c in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, and every expression *value* of type t in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, then
$$\mathsf{Update}(at_{(c,t)})[self, value] \in \mathrm{Stm}(D, H, w).$$
- For every association-end read action $\mathsf{Read}(as_{(c,c')}) \in \mathrm{Act}(D)$, every expression *self* of type c in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, and every *variable* of type $\mathrm{Set}(c')$ in $\mathrm{Var}^\sharp(H, w_{wt})$, then
$$\mathsf{Read}(as_{(c,c')})[self, variable] \in \mathrm{Stm}(D, H, w).$$
- For every association-end create action $\mathsf{Create}(as_{(c,c')}) \in \mathrm{Act}(D)$, every expression *self* of type c in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, and every expression *target* of type c' in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, then
$$\mathsf{Create}(as_{(c,c')})[self, target] \in \mathrm{Stm}(D, H, w).$$
- For every association-end delete action $\mathsf{Delete}(as_{(c,c')}) \in \mathrm{Act}(D)$, every expression *self* of type c in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, and every expression *target* of type c' in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, then
$$\mathsf{Delete}(as_{(c,c')})[self, target] \in \mathrm{Stm}(D, H, w).$$

Base case (GUI actions):

- For every type $t \in \mathrm{Typ}(D)$, every *variable* of type t in $\mathrm{Var}^\sharp(H, w_{wt})$ and every expression *value* of type t in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, then

$$\mathsf{Set}[\mathit{variable}, \mathit{value}] \in \mathrm{Stm}(D, H, w).$$

- For every window $\langle w', \mathsf{Window}\rangle \in W$, every list of variables $\mathit{variable}_1, \ldots,$ $\mathit{variable}_n$, such that, for $1 \leq i \leq n$, $\mathit{variable}_i$ is of type t_i in $\mathrm{Var}(H, w'_{\mathsf{Window}})$, and every list of expressions $\mathit{value}_1, \ldots, \mathit{value}_n$, such that, for $1 \leq i \leq n$, value_i is of type t_i in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, then

$$\mathsf{Open}[w', (\boldsymbol{\mathit{variable}, \mathit{value}})] \in \mathrm{Stm}(D, H, w).$$

- Finally,

$$\mathsf{Back}, \mathsf{Fail}, \mathsf{Skip} \in \mathrm{Stm}(D, H, w).$$

Inductive case

- For every expression *cond* of type $\mathsf{Boolean}$ in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, and every statements $\mathit{stm}_1, \mathit{stm}_2 \in \mathrm{Stm}(D, H, w)$, then

$$\mathsf{if\_then\_else}[\mathit{cond}, \mathit{stm}_1, \mathit{stm}_2] \in \mathrm{Stm}(D, H, w).$$

- For every expression *source* of type $\mathsf{Sequence}(t)$ in $\mathrm{Expr}(D, \mathrm{Var}^\sharp(H, w_{wt}))$, every widget variable *variable* of type t in $\mathrm{Var}^\sharp(H, w_{wt})$, and every statement $\mathit{body} \in \mathrm{Stm}(D, H, w)$, then

$$\mathsf{iterator}[\mathit{source}, \mathit{variable}, \mathit{body}] \in \mathrm{Stm}(D, H, w).$$

- For all statements stm, $\mathit{stm}' \in \mathrm{Stm}(D, H, w)$, then

$$\mathit{stm}\ ;\ \mathit{stm}' \in \mathrm{Stm}(D, H, w).$$

3.3 Behavioral GUIML models

Definition. Let D be a data model $\langle C, AT, AS, ASO\rangle$. Let H be a GUIML layout model $\langle W, WC, X, EV\rangle$. Then, a GUIML *behavioral model* G is a tuple $\langle D, H, EST\rangle$ such that:

- EST is a set of pairs $\langle\langle ev, w_{wt}\rangle, \mathit{stm}\rangle$, where
  - $\langle ev, w_{wt}\rangle \in EV$ is an event.
  - $\mathit{stm} \in \mathrm{Stm}(D, H, \mathrm{Win}(H, w_{wt}))$ is the statement associated to this event.

Invariants.

- Every event is associated with exactly one statement.
- In every sequence of statement associated to an event, the GUI actions Open and Back can only appear (if at all) at the last position.[5]

[5] When this last position is occupied by an if-then-else, then Open and Back can only appear (if at all) at the last position of its then- or else-branches (and recursively in the case of nested if-then-elses). The situation is similar for iterator statements.

3.4 Operational semantics for events

Notation. Let D be a data model $\langle C, AT, AS, ASO\rangle$. Let I be a D-object model $\langle O, VA, LK\rangle$. In what follows we will use the following notation:

- Let $o_c \in O$ be an object. Then, $(VA \setminus o_c)$ denotes the set that results from *deleting* from VA every triple that contains o_c. That is, $(VA \setminus o_c) = \{\langle o'_{c'}, at_{(c',t')}, va\rangle \mid \langle o'_{c'}, at_{(c',t')}, va\rangle \in VA \wedge o'_{c'} \neq o_c\}$.
- Let $o_c \in O$ be an object. Then, $(LK \setminus o_c)$ denotes the set that results from *deleting* from LK every triple that contains o_c. That is, $(LK \setminus o_c) = \{\langle o'_{c'}, as_{(c',c'')}, o''_{c''}\rangle \mid \langle o'_{c'}, as_{(c',c'')}, o''_{c''}\rangle \in LK \wedge o'_{c'} \neq o_c \wedge o''_{c''} \neq o_c\}$.
- Let $at_{(c,t)} \in AT$ be an attribute. Let $o_c \in O$ be an object and let $va \in [\![t]\!]^{\mathrm{TP}}$ be a value of type t. Then $VA \oplus \langle o_c, at_{(c,t)}, va\rangle$ denotes the set that results from *overriding* (i.e., updating) in VA the value of the attribute at of the object o_c with va. That is, $(VA \oplus \langle o_c, at_{(c,t)}, va\rangle) = \{\langle o_c, at_{(c,t)}, va\rangle\} \cup \{\langle o'_{c'}, at'_{(c',t')}, va'\rangle \mid \langle o'_{c'}, at'_{(c',t')}, va'\rangle \in VA \wedge o'_{c'} \neq o_c \wedge at'_{(c',t')} \neq at_{(c,t)}\}$.

Definition. Let D be a data model $\langle C, AT, AS, ASO\rangle$. Let H be a GUIML layout model $\langle W, WC, X, EV\rangle$. Let G be a GUIML behavioral model $\langle D, H, EST\rangle$. Let $ev \in EV$ be an event in G with $\langle ev, stm\rangle \in EST$. Then, $\mathrm{Sem}(G, ev)$ is the set of all the transitions

$$\langle stm, I, \theta\rangle \longrightarrow^* \langle \mathsf{Skip}, I', \theta'\rangle$$

where $\longrightarrow^*$ is the transitive closure of the small-step transition relation $\longrightarrow$ defined by the following inference rules. For every D-object model $I = \langle O, VA, LK\rangle$ and every (X, I)-substitution we have:

Base case (data actions)

$$\frac{o_c \notin O}{\langle \mathsf{Create}(c)[variable], I, \theta\rangle \longrightarrow \langle \mathsf{Skip}, \langle O \cup \{o_c\}, VA, LK\rangle, \theta \oplus \{variable \mapsto o_c\}\rangle}.$$

$$\frac{[\![(self)\theta]\!]^I = o}{\langle \mathsf{Delete}(c)[self], I, \theta\rangle \longrightarrow \langle \mathsf{Skip}, \langle (O \setminus o_c), (VA \setminus o_c), (LK \setminus o_c)\rangle, \theta\rangle}.$$

$$\frac{[\![(self.at)\theta]\!]^I = va}{\langle \mathsf{Read}(at_{(c,t)})[self, variable], I, \theta\rangle \longrightarrow \langle \mathsf{Skip}, I, \theta \oplus \{variable \mapsto va\}\rangle}.$$

$$\frac{[\![(self)\theta]\!]^I = o, [\![(value)\theta]\!]^I = va}{\langle \mathsf{Update}(at_{(c,t)})[self, value], I, \theta\rangle \longrightarrow \langle \mathsf{Skip}, \langle O, (VA \oplus \langle o, at, va\rangle), LK\rangle, \theta\rangle}.$$

$$\frac{[\![(self.as)\theta]\!]^I = \{o_1, \ldots, o_n\}}{\langle \mathsf{Read}(as_{(c,c')})[self, variable], I, \theta\rangle \longrightarrow \langle \mathsf{Skip}, I, \theta \oplus \{variable \mapsto \{o_1, \ldots, o_n\}\}\rangle}.$$

$$\frac{[\![(self)\theta]\!]^I = o, [\![(target)\theta]\!]^I = o', (as_{(c,c')}, as'_{(c',c)}) \in ASO}{\langle \mathsf{Create}(as_{(c,c')})[self, target], I, \theta\rangle \longrightarrow \langle \mathsf{Skip}, \langle O, VA, (LK \cup \{\langle o, as, o'\rangle, \langle o', as', o\rangle\})\rangle, \theta\rangle}.$$

$$\frac{[\![(self)\theta]\!]^I = o, [\![(target)\theta]\!]^I = o', (as_{(c,c')}, as'_{(c',c)}) \in ASO}{\langle \mathsf{Delete}(as_{(c,c')})[self, target], I, \theta\rangle \longrightarrow \langle \mathsf{Skip}, \langle O, VA, (LK \setminus \{\langle o, as, o'\rangle, \langle o', as', o\rangle\})\rangle, \theta\rangle}.$$

Base case (GUI actions)

$$\frac{[\![(value)\theta]\!]^I = va}{\langle \mathsf{Set}[variable, value], I, \theta\rangle \longrightarrow \langle \mathsf{Skip}, I, \theta \oplus \{variable \mapsto va\}\rangle}.$$

$$\frac{}{\langle \mathsf{Open}[\langle w, \mathsf{Window}\rangle, (\boldsymbol{variable, value})], I, \theta\rangle \longrightarrow \langle \mathsf{Skip}, I, \theta\rangle}.$$

$$\frac{}{\langle \mathsf{Back}, I, \theta\rangle \longrightarrow \langle \mathsf{Skip}, I, \theta\rangle}.$$

Inductive case

$$\frac{[\![(cond)\theta]\!]^I = \mathsf{true}}{\langle \mathsf{If\_then\_else}[cond, stm_1, stm_2], I, \theta\rangle \longrightarrow \langle stm_1, I, \theta\rangle}.$$

$$\frac{[\![(cond)\theta]\!]^I = \mathsf{false}}{\langle \mathsf{If\_then\_else}[cond, stm_1, stm_2], I, \theta\rangle \longrightarrow \langle stm_2, I, \theta\rangle}.$$

$$\frac{[\![(source)\theta]\!]^I = [v_1, \ldots, v_n]}{\langle \mathsf{Iterator}[source, variable, body], I, \theta\rangle \longrightarrow \langle (\mathsf{Set}(variable, v_1)\,;\, body\,;\ldots;\mathsf{Set}(variable, v_n)\,;\, body), I, \theta\rangle}.$$

$$\frac{\langle stm_1, I, \theta\rangle \longrightarrow \langle stm'_1, I', \theta'\rangle}{\langle (stm_1\,;\, stm_2), I, \theta\rangle \longrightarrow \langle stm'_1\,;\, stm_2, I', \theta'\rangle}.$$

$$\frac{}{\langle (\mathsf{Skip}\,;\, stm_2), I, \theta\rangle \longrightarrow \langle stm_2, I, \theta\rangle}.$$

4 Security-aware GUIML

In this section we first characterize *security-awareness* of GUIML behavior models in terms of a transition relation defined by a security-aware version of the inference rules that define the (non security-aware) operational semantics of GUIML events. Then, we define a model transformation that, given a GUIML model G and a SecureUML model S, generates a new GUIML model that is *security aware* with respect to S. Finally, we formalize and prove the correctness of our model transformation.

4.1 Operational semantics for security-aware events

Informally, security-aware events are those events where the execution of the associated actions are *conditional* on the satisfaction of the corresponding authorization constraints. However, which constraint these are depends, of course, on the role of the actual user who triggers this event. Thus, in order to be able to refer to the user's role (when specifying the aforementioned conditions within the statement associated to the event), we will explicitly require that:

- The underlying data model D includes a class Role, with an attribute name of type String.
- Every window in the GUIML model owns two distinguished variables, caller (of the same type than the users) and role (of type Role), whose intented values are, respectively, the actual user and its role

Moreover, when discussing security-awareness with respect to a security model S we will be interested only in D-object models whose objects of type Role are *conformant* with the roles declared in S, in the following sense: Let D be a data model $\langle C, AT, AS, ASO\rangle$, such that $\mathsf{Role} \in C$ and $\langle \mathsf{name}, \mathsf{Role}, \mathsf{String}\rangle \in AT$. Let S be a D-security model $\langle D, R, RH, u, P\rangle$. Let $I = \langle O, VA, LK\rangle$ be a D-object model. Then, we say that I is *R-conformant* if and only if

$$R = \{va \mid \langle o_{\mathsf{Role}}, \mathsf{name}_{(\mathsf{Role},\mathsf{String})}, va\rangle \in VA\}.$$

Definition. Let D be a data model $\langle C, AT, AS, ASO\rangle$, such that $\mathsf{Role} \in C$ and $\langle \mathsf{name}, \mathsf{Role}, \mathsf{String}\rangle \in AT$. Let S be a D-security model $\langle D, R, RH, u, P\rangle$. Let H be a GUIML layout model $\langle W, WC, X, EV\rangle$ such that, for every $w_{\mathsf{Window}} \in W$, $\langle \mathsf{role}_{\mathsf{Role}}, w_{\mathsf{Window}}\rangle \in X$ and $\langle \mathsf{caller}_u, w_{\mathsf{Window}}\rangle \in X$. Let G be a GUIML behavioral model $\langle D, H, EST\rangle$. Let $ev \in EV$ be an event in G whose associated statement is stm, i.e., $\langle ev, stm\rangle \in EST$. Then, the *security-aware* operational semantics for the event ev is given by the set of all the transitions

$$\langle stm, I, \theta\rangle \longrightarrow^*_{\mathrm{sec}} \langle \mathsf{Skip}, I', \theta'\rangle$$

such that I is R-conformant and $\longrightarrow^*_{\mathrm{sec}}$ is the transitive closure of the small-step transition relation $\longrightarrow_{\mathrm{sec}}$ defined by the *security-aware versions* of the inference rules that define the operational semantics of GUIML events. Formally, for every

GUIML data action $act[\boldsymbol{arg}]$, the security-aware version of the corresponding inference rule includes the following additional condition:

$$[\![(\mathrm{AuthPerm}(S, [\![(\mathsf{role}_{\mathrm{Win}(H,ev)}.\mathsf{name})\theta]\!]^I, act)(\mathrm{Subst}(act[\boldsymbol{arg}])))\theta]\!]^I = \mathsf{true},$$

where $\mathrm{Subst}(act[\boldsymbol{arg}])$ is the substitution defined below, which depends on the type of the action act.

$$\begin{aligned}
&\mathrm{Subst}(\mathsf{Create}(c)[variable]) = \\
&\qquad \{\mathsf{caller}_u \mapsto \langle \mathrm{caller}_u, w_{\mathsf{Window}}\rangle\}. \\
&\mathrm{Subst}(\mathsf{Delete}(c)[self]) = \\
&\qquad \{\mathsf{caller}_u \mapsto \langle \mathrm{caller}_u, w_{\mathsf{Window}}\rangle, \mathsf{self}_c \mapsto self\}. \\
&\mathrm{Subst}(\mathsf{Read}(at_{(c,t)})[self, variable]) = \\
&\qquad \{\mathsf{caller}_u \mapsto \langle \mathrm{caller}_u, w_{\mathsf{Window}}\rangle, \mathsf{self}_c \mapsto self\}. \\
&\mathrm{Subst}(\mathsf{Update}(at_{(c,t)})[self, value]) = \\
&\qquad \{\mathsf{caller}_u \mapsto \langle \mathrm{caller}_u, w_{\mathsf{Window}}\rangle, \mathsf{self}_c \mapsto self, \mathsf{value}_t \mapsto value\}. \\
&\mathrm{Subst}(\mathsf{Read}(as_{(c,c')})[self, variable]) = \\
&\qquad \{\mathsf{caller}_u \mapsto \langle \mathrm{caller}_u, w_{\mathsf{Window}}\rangle, \mathsf{self}_c \mapsto object\}. \\
&\mathrm{Subst}(\mathsf{Create}(as_{(c,c')})[self, target]) = \\
&\qquad \{\mathsf{caller}_u \mapsto \langle \mathrm{caller}_u, w_{\mathsf{Window}}\rangle, \mathsf{self}_c \mapsto self, \mathsf{target}_{c'} \mapsto target\}. \\
&\mathrm{Subst}(\mathsf{Delete}(as_{(c,c')})[self, target]) = \\
&\qquad \{\mathsf{caller}_u \mapsto \langle \mathrm{caller}_u, w_{\mathsf{Window}}\rangle, \mathsf{self}_c \mapsto self, \mathsf{target}_{c'} \mapsto target\}.
\end{aligned}$$

The inference rules for GUI actions are not modified in their security-aware versions. The inference rules for if-then-else statements, iterator statements, or sequences of statements also remain unmodified.

4.2 Security-aware model transformation

Definition. Let D be a data model $\langle C, AT, AS, ASO\rangle$, such that $\mathsf{Role} \in C$ and $\langle \mathsf{name}, \mathsf{Role}, \mathsf{String}\rangle \in AT$. Let S be a D-security model $\langle D, R, RH, u, P\rangle$. Let H be a GUIML layout model $\langle W, WC, X, EV\rangle$ such that, for every $w_{\mathsf{Window}} \in W$, $\langle \mathsf{role}_{\mathsf{Role}}, w_{\mathsf{Window}}\rangle \in X$ and $\langle \mathsf{caller}_u, w_{\mathsf{Window}}\rangle \in X$. Let G be a GUIML behavioral model $\langle D, H, EST\rangle$. Then, $\mathrm{Sec}(G, S)$ is the *S-security-aware version* of G, defined as follows:

$$\mathrm{Sec}(G, S) = \langle D, H, \{\langle ev, \mathrm{Sec}(stm, S)\rangle \mid \langle ev, stm\rangle \in EST\}\rangle.$$

Here $\mathrm{Sec}(stm, S)$ is the *S-security-aware version* of the statement stm associated to the event ev, defined recursively as follows:

Base case (data actions): Let $R = \{r_1, \ldots, r_n\}$. Then,

$$
\begin{array}{l}
\mathrm{Sec}(act[\boldsymbol{arg}], S) = \\
\quad \mathsf{If\_then\_else}[r_1 = \mathsf{role}_{\mathrm{Win}(H,ev)}.\mathsf{name}, \\
\qquad \mathsf{If\_then\_else}[\mathrm{AuthPerm}(S, r_1, act)(\mathrm{Subst}(act[\boldsymbol{arg}])), \\
\qquad\qquad act[\boldsymbol{arg}], \\
\qquad\qquad \mathsf{Fail}], \\
\qquad \ldots \\
\qquad\quad \mathsf{If\_then\_else}[r_n = \mathsf{role}_{\mathrm{Win}(H,ev)}.\mathsf{name}, \\
\qquad\qquad \mathsf{If\_then\_else}[\mathrm{AuthPerm}(S, r_n, act)(\mathrm{Subst}(act[\boldsymbol{arg}])), \\
\qquad\qquad\qquad act[\boldsymbol{arg}]), \\
\qquad\qquad\qquad \mathsf{Fail}], \\
\qquad\qquad \mathsf{Fail}] \ \ldots\ ].
\end{array}
$$

Here $\mathrm{Subst}(act[\boldsymbol{arg}])$ is the substitution defined above, where w_{Window} is in this case the window that contains the widget that supports the event ev.

Base case (GUI actions):

$\mathrm{Sec}(\mathsf{Set}[variable, value], S) = \mathsf{Set}[variable, value]$.
$\mathrm{Sec}(\mathsf{Back}, S) = \mathsf{Back}$.
$\mathrm{Sec}(\mathsf{Skip}, S) = \mathsf{Skip}$.
$\mathrm{Sec}(\mathsf{Open}[w_{\mathsf{Window}}, (\boldsymbol{variable}, \boldsymbol{value})], S) = \mathsf{Open}(w_{\mathsf{Window}}, (\boldsymbol{variable}, \boldsymbol{value}))$.

Inductive cases.

$$
\begin{array}{l}
\mathrm{Sec}(\mathsf{if\_then\_else}[cond, stm_1, stm_2], S) = \\
\quad \mathsf{if\_then\_else}[cond, \mathrm{Sec}(stm_1, S), \mathrm{Sec}(stm_2, S)]. \\
\mathrm{Sec}(\mathsf{iterator}[source, variable, body], S) = \mathsf{iterator}[source, variable, \mathrm{Sec}(body, S)]. \\
\mathrm{Sec}((stm_1\,;\, stm_2), S) = (\mathrm{Sec}(stm_1, S)\,;\, \mathrm{Sec}(stm_2, S)).
\end{array}
$$

4.3 Correctness

The following theorem basically states that the evaluation of a transformed statement following the non-security-aware operational semantics for events returns the same result than its evaluation using the security-aware version of this semantics and, therefore, that the transformed statement respects the authorization constraints formalized in the underlying security model.

Theorem. Let D be a data model $\langle C, AT, AS, ASO\rangle$, such that $\mathsf{Role} \in C$ and $\langle \mathsf{name}, \mathsf{Role}, \mathsf{String}\rangle \in AT$. Let S be a D-security model $\langle D, R, RH, u, P\rangle$. Let H be a GUIML layout model $\langle W, WC, X, EV\rangle$ such that for every $w_{\mathsf{Window}} \in W$,

$\langle \mathsf{role}_{\mathsf{Role}}, w_{\mathsf{Window}} \rangle \in X$ and $\langle \mathsf{caller}_u, w_{\mathsf{Window}} \rangle \in X$. Let G be a GUIML behavioral model $\langle D, H, EST \rangle$. Let $w_{\mathsf{Window}} \in W$ be a window and let $stm \in \mathrm{Stm}(D, H, w)$. Then, for every R-conformant D-object data model I, and every (X, I)-substitution θ,

$$\langle \mathrm{Sec}(stm, S), I, \theta \rangle \longrightarrow^* \langle \mathsf{Skip}, I', \theta' \rangle. \iff \langle stm, I, \theta \rangle \longrightarrow^*_{\mathrm{sec}} \langle \mathsf{Skip}, I', \theta' \rangle.$$

Proof. By induction on stm.

Acknowledgements

This work is partially supported by the EU FP7-ICT Project "NESSoS: Network of Excellence on Engineering Secure Future Internet Software Services and Systems" (256980) by the Spanish Ministry of Science and Innovation Project "DESAFIOS-10" (TIN2009-14599-C03-01), and by Comunidad de Madrid Program "PROMETIDOS-CM" (S2009TIC-1465).

References

1. Object Management Group. *Object Constraint Language specification Version 2.3.1*, January 2012. `http://www.omg.org/spec/OCL/2.3.1`.

Model-Driven Development of Security-Aware GUIs for Data-Centric Applications

David Basin[1], Manuel Clavel[2,3], Marina Egea[2], Miguel A. García de Dios[2], Carolina Dania[2], Gonzalo Ortiz[2], and Javier Valdazo[2]

[1] ETH Zürich, Switzerland
basin@inf.ethz.ch
[2] IMDEA Software Institute, Madrid, Spain
{manuel.clavel,marina.egea,miguelangel.garcia}@imdea.org,
{carolina.dania,gonzalo.ortiz,javier.valdazo}@imdea.org
[3] Universidad Complutense, Madrid, Spain
clavel@sip.ucm.es

Abstract. In this tutorial we survey a very promising instance of model-driven security: the full generation of security-aware graphical user interfaces (GUIs) from models for data-centric applications with access control policies. We describe the modeling concepts and languages employed and how model transformation can be used to automatically lift security policies from data models to GUI models. We work through a case study where we generate a security-aware GUI for a chatroom application. We also present a toolkit that supports the construction of security, data, and GUI models and generates complete, deployable, web applications from these models.

1 Introduction

Model building is at the heart of system design. This is true in many engineering disciplines and is increasingly the case in software engineering. Model-driven engineering (MDE) [7] is a software development methodology that focuses on creating models of different system views from which system artifacts such as code and configuration data are automatically generated. Proponents of model-driven engineering have in the past been guilty of making overambitious claims: positioning it as the Holy Grail of software engineering where modeling completely replaces programming. This vision is, of course, unrealizable in its entirety for simple complexity-theoretic reasons. If the modeling languages are sufficiently expressive then basic problems such as the consistency of the different models or views of a system become undecidable. However, there are specialized domains where MDE can truly deliver its full potential: in our opinion, security-aware GUIs for data-centric applications is one of them.

Data-centric applications are applications that manage information, typically stored in a database. In many cases, users access this information through graphical user interfaces (GUIs). Informally, a GUI consists of widgets (e.g., windows, text-fields, lists, and combo-boxes), which are visual elements that display and

A. Aldini and R. Gorrieri (Eds.): FOSAD VI, LNCS 6858, pp. 101–124, 2011.

store information and support events (like "clicking-on" or "typing-in"). A GUI defines the layout for the widgets, as well as the actions that the widgets' events trigger either on the application's database (e.g., to create, delete, or update information) or upon other widgets (e.g., to open or close a window).

There is an important, but little explored, link between visualization and security: When the application data is protected by an access control policy, the application GUI should be aware of and respect this policy. For example, the GUI should not display options to users for actions (e.g., to read or update information) that they are not authorized to execute on application data. This, of course, prevents the users from getting (often cryptic) security warnings or error messages directly from the database management system. It also prevents user frustration, for example from filling out a long electronic form only to have the server reject it because the user lacks a permission to execute some associated action on the application data. However, manual encoding the application's security policy within the GUI code is cumbersome and error prone. Moreover, the resulting code is difficult to maintain, since any changes in the security policy will require manual changes to the GUI code.

In this tutorial we spell out our model-driven engineering approach for developing security-aware GUIs for data-centric applications. The backbone of this approach, illustrated in Figure 1, is a model transformation that automatically lifts the access control policy modeled at the level of the data to the level of the GUI [2]. More precisely, given a security model (specifying the access control policy on the application data) and a GUI model (specifying the actions triggered by the events supported by the GUI's widgets), our model transformation generates a GUI model that is security-aware. The key idea underlying this transformation is that the link between visualization and security is ultimately defined in terms of data actions, since data actions are both controlled by the security policy and triggered by the events supported by the GUI. Thus, under our approach, the process of modeling and generating security-aware GUI has the following parts:

1. Software engineers specify the application-data model.
2. Security engineers specify the security-design model.
3. GUI designers specify the application GUI model.
4. A model transformation automatically generates a security-aware GUI model from the security model and the GUI model.
5. A code generator automatically produces a security-aware GUI from the security-aware model.

The other key components of this approach are the languages that we propose for modeling the data (ComponentUML), the access control policy (SecureUML), and the GUI (ActionGUI). These languages are defined by their corresponding metamodels and support the rigorous modeling of a large class of data models, security models, and GUI models. For data models, the main modeling elements are entities, along with their attributes and associations; for security models,

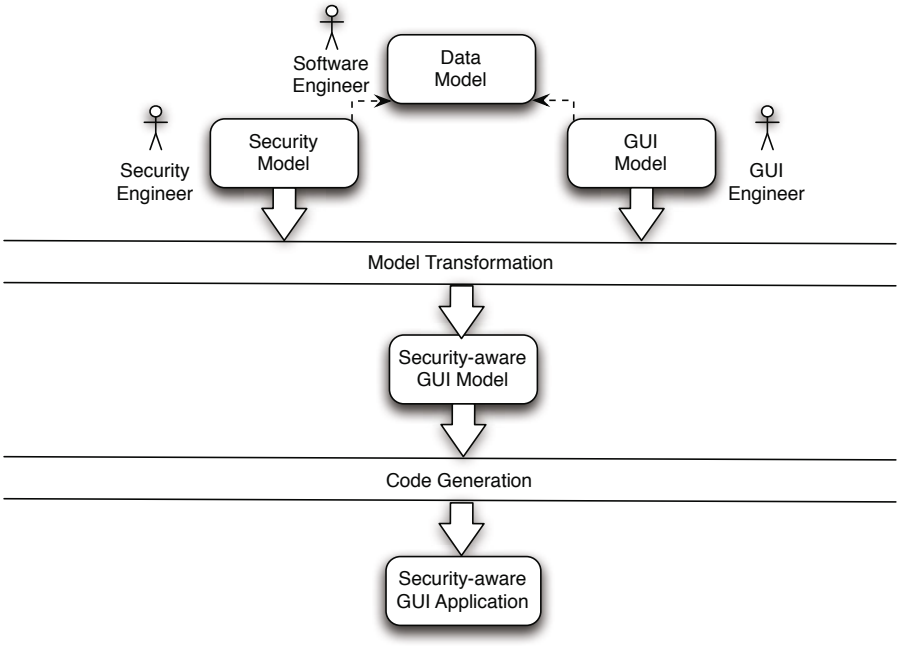


Fig. 1. Model-driven development of security-aware GUIs

these elements are roles, permissions (possibly constrained at runtime to satisfy given properties), and the actions associated to these permissions. For GUI models, these elements are widgets, the (possibly conditional) events associated to these widgets, and the (possibly conditional) actions associated to these events. The constraint language OCL [10] is used in all of these models. For security models, OCL is used to formalize the constraints on the permissions. For GUI models, it is used to formalize the conditions on the actions, as well as to specify the information to be displayed in widgets, updated in the database, or passed from one widget to another.

To support a full model-driven engineering development process, we have built a toolkit, named the ActionGUI Toolkit. This features specialized model editors for data, security, and GUI models, and implements the aforementioned model transformation to automatically generate security-aware GUI models. Moreover, our toolkit includes a code generator that, given a security-aware GUI model, automatically produces a complete web application, ready to be deployed in web containers such as Tomcat or GlassFish. A key component of this code generator is our translator from OCL to an SQL-based query language [5], which handles the OCL expressions appearing in the security-aware GUI models. More information about our ActionGUI Toolkit can be found at the URL `http://www.bm1software.com/actiongui.html`.

Overall, we see the full generation of security-aware GUIs from models for data-centric applications as a very promising application for model-driven engineering. By working with models and using code-generators to produce the final products, GUI designers can focus on the GUI's layout and behavior, instead of wrestling with the different, often complex, technologies that are used to implement them. Moreover, by using model transformations, the problem of establishing the link between visualization and security is successfully addressed.

To appreciate this last point, consider the standard alternative: the default, "ad-hoc" solution of directly hard-coding the security policy within the GUI. This is clearly disadvantageous. First, the GUI designer is often unaware of the application data security policy. Second, even if the designer is aware of it, manual hard-coding the application data security policy within the GUI code is cumbersome and error-prone. Finally, any changes in the security policy will require manual changes to the GUI code that implements this policy, which again is cumbersome and error-prone.

Organization

We explain in this tutorial our approach for developing security-aware GUIs for data-centric applications and present a toolkit, named ActionGUI, supporting this approach. We begin in Sections 2–4 by introducing our modeling languages for data models (ComponentUML), security models (SecureUML), and GUI models (ActionGUI). We also introduce our running example: a basic chatroom application. In Section 5, we discuss the problem of lifting the security requirements from data models to GUI models, and we present our solution: a model transformation that automatically transforms a GUI model into a security-aware GUI model with respect to the security requirements imposed on the underlying data model. We conclude this tutorial with a discussion on current and future work. All of the models we present here (and many more) are available at the ActionGUI home page. The interested reader can also evaluate there the code generated by the ActionGUI Toolkit from these models.

2 Data Models: ComponentUML

In this and the next two sections, we introduce the modeling languages that we use for the model-driven development of security-aware GUIs for data centric applications. These languages are: ComponentUML, for modeling data; SecureUML, for modeling the access control policy; and ActionGUI, for modeling the application's GUI. To illustrate the main modeling concepts and relationships provided by these languages, we work through a running example: a simple chat application named ChitChat, which supports multiple chat rooms where users can converse with each other in different chat rooms. We begin

then with ComponentUML, which is the language that we use for modeling the application data. ComponentUML also gives us a context to introduce the constraint language OCL [10], which we intensively use in our approach when modeling both access control policies and the application's GUIs.

Data models provide a data-oriented view of a system. Typically they are used to specify how data is structured, the format of data items, and their logical organization, i.e., how data items are grouped and related. ComponentUML essentially provides a subset of UML class models where *entities* (classes) can be related by *associations* and may have *attributes* and *methods*. Attributes and *association-ends* have *types* (either primitive types or entity types). As expected, the type associated to an attribute is the type of the attribute's values, and the type associated to an association-end is the type of the objects which may be linked at this end of the association. The ComponentUML metamodel is shown in Figure 2.

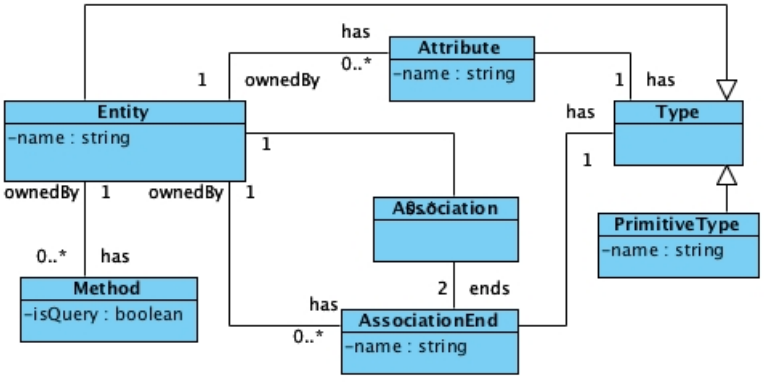


Fig. 2. The ComponentUML metamodel

The ChitChat data model. In our ChitChat application, each user has a nickname, a password, an e-mail address, a mood message, and a status. The user may participate and be invited to participate in any number of chat rooms. Each chat room is created by a user, has a name, a starting and ending date, and it manages the messages sent by its participants.

The table shown in Figure 3 specifies the data model of the ChitChat application, using the concepts and relationships provided by ComponentUML. Each row in this table corresponds to an entity, and shows its attributes and association-ends, with their corresponding types. Notice that we also indicate, for each association-end, the association-end which corresponds to the other end of the association. The reader can find on the ActionGUI home page the graphical representation of the ChitChat data model using the concrete syntax for ComponentUML that is supported by the ActionGUI Toolkit.

Entity	Attribute	Type	AssocEnd	Type	*Other end*
ChatUser	nickname	String	msgSent	ChatMessage	*from*
	password	String	participates	ChatRoom	*participants*
	email	String	owns	ChatRoom	*ownedBy*
	moodMsg	String	invitedTo	ChatRoom	*invitees*
	status	String			
ChatRoom	name	String	messages	ChatMessage	*chat*
	start	Date	participants	ChatUser	*participates*
	end	Date	ownedBy	ChatUser	*owns*
			invitees	ChatUser	*invitedTo*
ChatMessage	body	String	from	ChatUser	*msgSent*
			chat	ChatRoom	*messages*

Fig. 3. The ChitChat data specification

OCL: constraints and queries

The Object Constraint Language (OCL) [10] is a specification language for expressing constraints and queries using a textual notation. As part of the UML standard, it was originally intended for modeling properties that could not be easily or naturally captured using graphical notation (e.g., class invariants in a UML class diagram). In fact, OCL expressions are always written in the context of a model, and they are evaluated on an instance of this model. This evaluation returns a value but does not change anything; OCL is a side-effect free language.

We summarize here the main elements of the OCL language which are used in this tutorial. OCL is a strongly type language. Expressions either have a primitive type (namely, Boolean, Integer, Real, and String), a class type, or a collection type, whose base type is either a primitive type or a class type. OCL provides the standard operators on primitive types and on collections. For example, the operator includes checks whether an object is part of a collection, and the operator isEmpty checks whether a collection is empty. More interestingly, OCL provides a dot-operator to access the values of the objects' attributes and association-ends. For example, let u be an object of the class ChatUser. Then, the expression u.nickname refers to the value of the attribute nickname for the ChatUser u, and the expression u.participates refers to the objects linked to the ChatUser u through the association-end participates. Furthermore, OCL provides the operator allInstances to access to all the objects of a class. For example, the expression ChatRoom.allInstances() refers to all the objects of the class ChatRoom. Finally, OCL provides operators to iterate on collections. These are forAll, exists, select, reject, and collect. For example, ChatUser.allInstances()−>select(u|u.status='on−line') refers to the collection of objects of the class ChatUsers whose attribute status has the value "on-line".

ChitChat's entity invariants. To illustrate the syntax (and the semantics) of the OCL language, we formalize here some entity (class) invariants for ChitChat's

data model. For example, the following OCL expression formalizes that users' nicknames must be unique:

```
ChatUser.allInstances()->forall(u1,u2| u1 <> u2 implies u1.nickname <> u2.nickname).
```

Similarly, we can formalize that the status of a ChitChat user is either "off-line" or "on-line" using the following OCL expression:

```
ChatUser.allInstances() ->forall(u|u.status='off-line' or u.status='on-line').
```

Finally, we can formalize that each message has exactly one sender:

```
ChatMessage.allInstances()->forAll(m|m.from->size()= 1).
```

3 Security Models: SecureUML+ComponentUML

SecureUML [3] is a modeling language for formalizing access control requirements that is based on RBAC [6]. In RBAC, permissions specify which roles are allowed to perform given operations. These roles typically represent job functions within an organization. Users are granted permissions by being assigned to the appropriate roles based on their competencies and responsibilities in the organization. RBAC additionally allows one to organize the roles in a hierarchy where roles can inherit permissions along the hierarchy. In this way, the security policy can be described in terms of the hierarchical structure of an organization. However, it is not possible in RBAC to specify policies that depend on dynamic properties of the system state, for example, to allow an operation only during weekdays. SecureUML extends RBAC with *authorization constraints* to overcome this limitation.

SecureUML provides a language for specifying access control policies for actions on protected resources. However, it leaves open what the protected resources are and which actions they offer to actors. These are specified in a so-called "dialect". Figure 4 shows the SecureUML metamodel. Essentially, it provides a language for modeling *roles* (with their hierarchies), *permissions*, *actions*, *resources*, and *authorization constraints*, along with their assignments, i.e., which permissions are assigned to a role, which actions are allowed by a permission, which resource is affected by the actions allowed by a permission, which constraints need to be satisfied for granting a permission, and, finally, which resource is affected by an action.

In our approach, we use a specific dialect of SecureUML, named SecureUML+ComponentUML, for modeling the access control policy on data models. The SecureUML+ComponentUML metamodel provides the connection between SecureUML and ComponentUML. Essentially, in this dialect of SecureUML, the protected resources are the entities, along with their attributes, methods, and association-ends (but not the associations as such), and the actions that they offer to the actors are those shown in Figure 5. Essentially, there are two classes of actions: atomic and composite. The *atomic* actions are intended to map directly

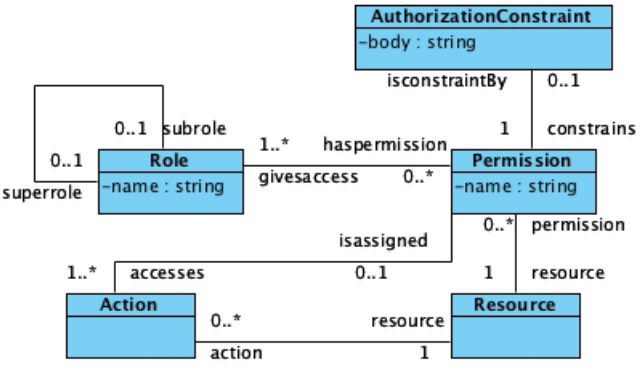


Fig. 4. The SecureUML metamodel

Resource	Actions
Entity	create, read, update, delete, full access
Attribute	read, update, full access
Method	execute
AssociationEnd	read, create, delete, full access

Fig. 5. The SecureUML+ComponentUML actions on protected resources

onto actual operations on the database. These actions are: create and delete for entities; read and update for attributes; create and delete for association-ends; and execute for methods. The underlined actions are the *composite* actions, which hierarchically group lower-level actions. Composite actions allow modelers to conveniently specify permissions for sets of actions. For example, the full access action for an attribute groups together the read and update actions for this attribute.

Finally, in SecureUML+ComponentUML, authorization constraints are specified using OCL, extended by four keywords, self, caller, value, and target. These keywords have the following meanings:

- self refers to the root resource upon which the action will be performed, if the permission is granted. The root resource of an attribute, an association-end, or a method is the entity to which it belongs.
- caller refers to the actor that will perform the action, if the permission is granted.
- value refers to the value that will be used to update an attribute, if the permission is granted.
- target refers to the object that will be linked at the end of an association, if the permission is granted.

The ChitChat access control policy. For the sake of our running example, consider the following (partial) access control policy for the ChitChat application:

- Only administrators can create or delete users;
- Administrators can read any user's nickname, email, mood message, and status.
- Any user can read and update its own nickname, password, email, mood message, and status.
- Any user can read other users' nicknames, mood messages, and status.
- Users can join a chat room by invitation only, but they can leave at any time.

The table shown in Figure 6 specifies this (partial) access control policy, using the concepts and relationships provided by SecureUML+ComponentUML. Each row in this table corresponds to a role, and shows its permissions. Moreover, for each permission it shows the actions allowed by the permission, the resource affected by each of these actions, and the constraint that must be satisfied for the permission to be granted. For example, the permission DisjointChat authorizes any user to leave a chat room at any anytime. More precisely, it allows any user caller to delete a participates-link between a user self and a chat room target (meaning that the user self leaves the chat room target), but only if the user caller is indeed the user self (that is, the user caller is the one leaving the chat room target), and also the chat room target indeed belongs to the collection of chat rooms linked to the user caller through the association-end participates (that is, the caller is actually participating in the chat room target). The reader can find in the ActionGUI home page the graphical representation of ChitChat's access control policy using the concrete syntax for SecureUML+ComponentUML that is supported by the ActionGUI Toolkit.

Role	**Permission**	**Action**	**Resource**	**Authorization Constraint**
Admin	AnyUser	create, delete	ChatUser	true
		read	nickname	
		read	email	
		read	moodMsg	
		read	status	
User	SelfUser	read, update	nickname	caller=self
		read, update	password	
		read, update	email	
		read, update	moodMsg	
		read, update	status	
	OtherUser	read	nickname	true
		read	moodMsg	
		read	status	
	JointChat	create	participates	self=caller and caller.invitedTo−>includes(target)
	DisjointChat	delete	participates	self=caller and caller.participates−>includes(target)

Fig. 6. The ChitChat access control policy (partial)

4 GUI Models: ActionGUI

ActionGUI is a modeling language for formalizing the GUIs of a rich class of data-centric applications. The ActionGUI metamodel is shown in Figure 7. In a nutshell, ActionGUI provides a language to model *widgets* (e.g., windows, text-fields, buttons, lists, and tables), *events* (e.g., clicking-on, typing-in), and *actions*, which can be on data (e.g., to update a property of an element in the database) or on other widgets, (e.g., to open a window), as well as the associations that link the widgets with the events that they support and the events with the actions that they trigger. In addition, ActionGUI provides support to formally model the following features:

- Widgets can be displayed in *containers*, which are also widgets (e.g., a window can contain other widgets).
- Widgets may own *variables*, which are in charge of storing information for later use.
- Events may be only supported upon the satisfaction of specific *conditions*, whose truth value can depend on the information stored in the widgets' variables or in the database.
- Actions may be only triggered upon the satisfaction of specific *conditions*, whose truth value can depend on the information stored in the widgets' variables or in the database.
- Actions may take their *arguments* (values that instantiate their parameters) from the information stored in the widgets' variables or in the database.

The ActionGUI metamodel's invariants specify: (i) for each type of widget, the "default" variables that widgets of this type always own; (ii) for each type of widget, the type of events that widgets of this type may support; and (iii) for each type of action, the arguments that actions of this type require, as well as the arguments (if any) that these actions may additionally take. In particular, the invariants of ActionGUI's metamodel formalize, among others, the following constraints about the different types of widgets:

- *Windows.* They can contain any type of widget, except windows. Windows are not contained in any widget.
- *Text-field.* They can be typed-in. By default, each text-field owns a variable text of type string, which stores the last string typed-in by the user. The value of the variable text is permanently displayed in the text-field.
- *Button.* They can be clicked-on.
- *List.* They contain exactly one text-field. By default, each list owns a variable rows of type collection. A list displays as many rows as elements are in the collection stored by the variable rows, each row containing exactly one instance of the text-field contained by the list. By default, each instance of this text-field owns a variable row whose value is the element associated to this row from those stored by the variable rows. Finally, by default, each list owns a variable selected that holds the element associated to the last row selected by the user.

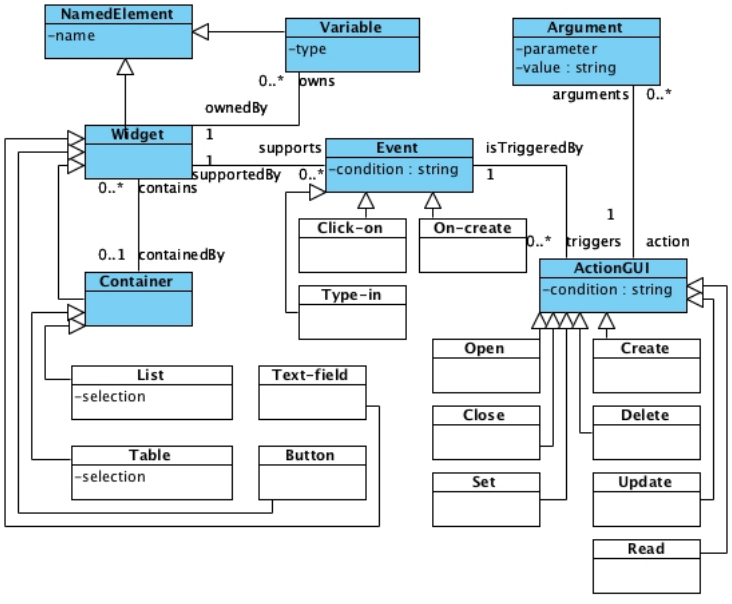


Fig. 7. The ActionGUI metamodel

- *Combo-box.* They are similar to lists, except that rows are displayed in a drop-down box.
- *Table.* They are similar to lists, except that they can contain any number of text-fields, buttons, lists, or even tables.

Also, the invariants of ActionGUI's metamodel formalize, among others, the following invariants about the different types of actions:

- *Create.* It creates a data item in the database. It takes two arguments: the type of the new data item (type) and the variable that will store this element for later reference (variable).
- *Delete (entities).* It deletes a data item in the database. It takes as argument the element to be deleted (object).
- *Read.* It reads the value of a data item's attribute in the database. It takes three arguments: the data item whose property is to be read (object); the property to be read (attribute); and the variable that will store, for later reference, the value read (variable).
- *Update.* It modifies the value of a data item's attribute in the database. It takes three arguments: the data item whose attribute is to be modified (object); the attribute to be modified (attribute); and the new value (value).
- *Create (association-ends).* It creates a new link in the database between two data items. It takes three arguments: the source data item (sourceObject); the target data item (targetObject); and the association-end (associationEnd) through which the target data item will be linked to the source data item.

- *Delete (association-ends).* It deletes a link in the database between two data items. It takes three arguments: the source data item (sourceObject); the target data item (targetObject); and the association-end (associationEnd) from where the target data item will be removed.
- *Open.* It opens a window. It takes as argument the window to be opened (target); additionally, for any of this window's variables, it can take as argument a value to be assigned to this variable when opening the window.
- *Back.* It goes back to the window from which a window was open.
- *Set.* It assigns a new value to a widget's variable. It takes two arguments: the variable (target) and the value to be assigned to this variable (value).

Finally, actions' conditions and arguments are specified in ActionGUI models using OCL, extended with the widget's variables (always enclosed in square brackets). As expected, when evaluating an OCL expression that contains a widget's variable, the value of the corresponding subexpression is the value currently stored in the variable. In case of ambiguity, a widget's variable is denoted by its name, prefixed by the name of its widget (followed by a dot). Also, in case of ambiguity, the name of a widget is prefixed by the name of its container (followed by a dot). Notice that, within the same containers, widgets have unique names. Moreover, a widget's variable can only be used within the window that contains its widget, either directly or indirectly.

The ChitChat login window. To continue with our running example, consider the following interface for allowing a registered user to login into the ChitChat application: a window (loginWi) containing:

- a writable text-field (nicknameEn), for the user to type its nickname in;
- a writable text-field (passwordEn), for the user to type its password in; and
- a clickable button (loginBu), for the user to login, using as its nickname and password the strings that it typed in the text-fields nicknameEn and passwordEn, respectively. Upon successful authentication, the user will be directed to the application's main menu window (menuWi) as the logged-in user (caller).

The table shown in Figure 8 specifies this login window, using the concepts and relationships provided by ActionGUI. Each row in this table correspond to a widget, where the containment relationship is denoted by displaying the widgets using tree-like notation. For each widget, we show the variables owned by the widget and the events that it supports. Moreover, for each event, we show the actions triggered by this event, as well as its arguments, indicating the values for each of the actions' parameters. However, we neither show in this table the "default" widget's variables nor the events supported by the widgets when they do not trigger any action.

Notice also that there are two elements that we have intentionally left undefined in this table: the value to be assigned to the menuWi's variable caller when opening the window menuWi, and the condition for this action. We now describe how both elements can be defined using ActionGUI's extension of OCL.

Widgets	Variables	Events	Actions	Arguments
loginWi				
|nicknameEn				
|passwordEn				
|loginBu | |		click-on	open(⋆)	window = menuWi menuWi.caller = *authenticated user*

(⋆) *Upon successful authentication.*

Fig. 8. The ChitChat login window

– The *authenticated user* should be the registered user in the database whose nickname and password coincide with the values of the text-variables owned, respectively, by text-fields nicknameEn and passwordEn. Using our extended OCL, we can define the *authenticated user* as follows:

```
ChatUser.allInstances()->any(u|
    u.nickname= [nicknameEn.text] and u.password=[passwordEn.text])
```

Notice that, as one of the invariants of the ChitChat data model, we specified that nicknames shall be unique. Thus, although the any-iterator will return *any* registered user satisfying the body of the any-iterator, there will be at most one such registered users.

– The condition for opening the window menuWi should be the existence in the database of a registered user whose nickname and password coincide with the values of the text-variables owned, respectively, by text-fields nicknameEn and passwordEn. We can define this condition as follows:

```
ChatUser.allInstances()->exists(u|
    u.nickname= [nicknameEn.text] and u.password=[passwordEn.text])
```

The reader can find in the ActionGUI home page the graphical representation of ChitChat's login window using the concrete syntax for ActionGUI that is supported by the ActionGUI Toolkit.

The ChitChat menu window. Consider now the following interface for allowing a logged-in user to choose an option from ChitChat's main menu: a window (menuWi), owning a variable caller which stores the logged-in user, and containing:

– a selectable list (usersLi) with as many rows as registered users are online, each of these rows containing an unwritable text-field (nicknameLa) showing the nickname of the registered user associated to this row;
– a clickable button (editProfileBu) for the caller to access the interface for editing the profile (i.e., name, password, email, mood message, and status) of the user selected in the list usersLi;

– a clickable button (createChatBu) for the caller to access the interface for creating a new chat room; and
– a clickable button (closeChatBu) for the caller to close the window.

The table shown in Figure 9 specifies this menu window, using the concepts and relationships provided by ActionGUI. Notice that the collection of data items to be displayed in the list `usersLi`, namely, the *online users*, is not formally defined in this table. Using ActionGUI's extension of OCL, we can define this collection as follows:

```
ChatUser.allInstances()->select(u|u.status= 'on-line')
```

The reader can find in the ActionGUI home page the graphical representation of ChitChat's menu window using the concrete syntax for ActionGUI that is supported by the ActionGUI Toolkit.

Widgets	Variables	Events	Actions	Arguments
menuWi	caller			
| usersLi		on-create	set	target = rows value = *on-line users*
| | nicknameLa		on-create	read	object = [UsersLi.row] attribute = nickname variable = text
| editProfileBu		click-on	open	window = editProfileWi editProfileWi.selectedUser = [UsersLi.selected]
| createChatBu		click-on	open	window = createChatWi
| closeChatBu		click-on	close	

Fig. 9. The ChitChat menu window

5 Security-Aware ActionGUI Models

In this section, we propose our solution to what we believe is the key challenge when modeling security-aware GUIs for data-centric applications: Which method should the GUI designer use for establishing the link between visualization and security? In other words, how should the GUI designer model their GUIs so as to make them *aware of* and *respect* the access control policy that protects the application data? As mentioned before, establishing this link is crucial when modeling security-aware GUIs for data-centric applications. Basically, a GUI should not display options to users for actions (e.g., to read or update information) that they are not authorized to execute on application data. This prevents the users from getting security warnings or cryptic error messages directly from the database management system. It also prevents user frustration from filling out a long electronic form only to have the server reject it because the user lacks a permission to execute some associated action on the application data.

As we will motivate in this section, *manually* modeling (and even worse, manually encoding) the application's security policy within the GUI model is cumbersome, error prone, and scales poorly to large applications. Moreover, the resulting models are difficult to maintain, since any changes in the security policy will require manual changes to the GUI models. Our solution to this problem uses a standard technique in model-driven development: *model transformation*. In particular, we introduce a model transformation that *automatically* transforms a GUI model into a security-aware GUI model with respect to the access control policy imposed on the underlying data model. One additional and substantial advantage of our solution is that, by keeping the security models and the GUI model apart, the security engineers and the GUI designers can independently model what they know best and maintain their models independently.

The ChitChat edit-profile window. To motivate the problem faced by a GUI designer when modeling a security-aware GUI, let us continue with our running example. Consider the interface for allowing a logged-in user (caller) to edit the profile of a previously chosen user (selectedUser), which may of course be the caller itself. More specifically, this interface shall consist of a window such that:

1. The current values of the selectedUser's profile are displayed when opening the window.
2. The caller can type in the new values (if any) for the selectedUser's profile.
3. The selectedUser's profile is updated with the new values typed in by the caller when he or she clicks on a designated button.

Recall that a registered user's profile is composed of the following attributes: nickname, password, mood message, email, and status. Recall also that the access control policy for reading and updating users' profiles, as specified in Figure 6, is the following:

4. A user is always allowed to read and update its own nickname, password, mood message, email, and status.
5. A user is allowed to read another user's nickname, mood message, and status, but not the user's password or email.
6. An administrator is always allowed to read a user's nickname, mood message, status, and email, but not the user's password.

Now, if the GUI designer only takes into consideration the functional requirements (1–3), the ChitChat edit-profile window can be modeled as shown in Figure 10. Namely, a window editProfileWi, which owns the variable selectedUser and caller, and contains:

– A writable text-field nicknameEn, for the caller to type in the new value (if any) with which to update the selectedUser's nickname. Notice that when the text-field nicknameEn is created, its "default" variable text will be assigned the current value of the selectedUser's nickname, and therefore this value will be the string initially displayed in the text-field nicknameEn, as requested.

Widgets	Variables	Events	Actions	Arguments
editProfileWi	caller, selectedUser			
| nicknameEn		on-create	read	object = [selectedUser] attribute = nickname variable = text
| passwordEn		on-create	read	object = [selectedUser] attribute = password variable = text
| moodMsgEn		on-create	read	object = [selectedUser] attribute = moodMsg variable = text
| emailEn		on-create	read	object = [selectedUser] attribute = email variable = text
| statusEn		on-create	read	object = [selectedUser] attribute = status variable = text
| updateBu		click-on	update	object = [selectedUser] attribute = nickname value = [nicknameEn.text]
			update	object = [selectedUser] attribute = password value = [passwordEn.text]
			update	object = [selectedUser] attribute = moodMsg value = [moodMsgEn.text]
			update	object = [selectedUser] attribute = email value = [emailEn.text]
			update	object = [selectedUser] attribute = status value = [statusEn.text]

Fig. 10. The ChitChat edit-profile window (although security-unaware)

- Analogous writable text-fields for each of the other elements in a registered user's profile: password, mood message, email, and status.
- A clickable button updateBu for the caller to trigger the sequence of actions that will update, as requested, the selectedUser's nickname, password, mood message, and status, with the new values (if any) typed by the caller in the corresponding text-fields.

Obviously, the edit-profile window modeled in Figure 10 does not satisfy the security requirements (4–6). Any caller can read and update any value contained in the profile of any selectedUser! So how can the GUI designer model the edit-profile window to make it aware of and respect the security requirements (4–6)?

There are essentially two solutions available. Unfortunately both of them, if applied *manually*, are cumbersome and error prone and therefore impractical for large applications with complex security policies. To understand the challenge that faces a GUI designer when modeling security-aware GUIs, let us first highlight the knowledge that he or she must acquire to accomplish this task. We decompose this into two steps.

Step 1: For each action triggered by an event, and for each role considered by the access control policy, the GUI designer must determine (i) *the conditions under which the given action can be securely executed by a user with the given role*. This depends, of course, on the underlying access control policy, and, more specifically, on the authorization constraints assigned to the permissions which grant access to execute the given action for the given role. Also, recall that permissions are inherited along the role hierarchy.

Notice however that, to obtain (i), the GUI designer can not simply compose, using disjunctions, the authorization constraints assigned to the aforementioned permissions. In particular, the variable caller, if it appears in any of these authorization constraints, must be replaced by the (widget) variable that stores the current application's user. Furthermore, the variables self, value, and target, if they appear in any of the aforementioned authorization constraints, must also be replaced by the appropriate expressions, based on the arguments taken by the given action. For example, in the case of a read-action, the variable self should be replaced by the value of the parameter object for this action.

Step 2: For each event supported by a widget, and for each role considered by the access control policy, the GUI designer must determine (ii) *the conditions under which all the actions triggered by the given event can be securely executed by a user with the given role*. In this case, for each given role, the GUI designer can simply compose, using conjunctions, the results to determine (i) for every action triggered by the given event.

Let us now illustrate, using our running example, the two solutions that are currently available to the GUI modeler for turning a security-unaware GUI model into a security-aware one. To simplify the discussion, we assume from now on that the user currently logged-in is always stored in a (widget) variable caller, that this variable is owned by every window, and that this variable's value is automatically passed from one window to another window when opening the latter from the former. Similarly, we assume the role of the user currently logged-in (if he or she has several roles, then the "active" role) is always stored in a (widget) variable role, that this variable is owned by every window, and that this variable's value is automatically passed from one window to another window when opening the latter from the former.

Solution A. The first solution for the GUI designer consists in modeling as many different edit-profile windows as possible security scenarios. In our case, the

GUI designer must model three different edit-profile windows (editMyProfileWi, editOthersProfile, and editUsersProfile), one for each of the following scenarios:

1. When the caller has the role 'User' and coincides with the selectedUser.
2. When the caller has the role 'User' but does not coincide with the selectedUser.
3. When the caller has the role 'Admin'.

In particular, the window editMyProfileWi associated to the security scenario (1) will contain exactly the same widgets as the window editProfileWi in Figure 10. In contrast, the window editOthersProfileWi associated to the security scenario (2), will only contain the text-fields nicknameEn, moodMsgEn, and statusEn (and not the button updateBu), since a user with the role 'User' can only read (but not update) other user's nickname, mood message, and status. Similarly, the window editProfilesWi will only contain the text-fields nicknameEn, moodMsgEn, emailEn, and statusEn (and not the button updateBu), since a user with the role 'Admin' can read (but not update) any user's profile, except its password.

Furthermore, for this solution to work, every event intended to give access to the interface for editing users' profiles must also be aware of the security scenarios (1)–(3), in order to open the appropriate edit-profile window. In particular, the GUI designer must associate the sequence of conditional actions shown in Figure 11 to each of the aforementioned events. Notice that for each open-action, the imposed condition formalizes *the conditions under which all the actions triggered by all the events supported by the widgets which are contained in the window being opened can be securely executed by a user*. That is, the GUI designer must gather all the knowledge corresponding to (ii), in Step 2 above, for every event supported by any widget contained in the window which the action under consideration is about to open.

Actions		**Arguments**
if		[role] = 'User' and [caller] = [selected_user]
then	open	window = editMyProfileWi editMyProfileWi.selectedUser = [selected_user]
if		[role] = 'User' and [caller] <> [selected_user]
then	open	window = editOthersProfileWi editOthersProfileWi.selectedUser = [selectedUser]
if		[role] = 'Admin'
then	open	window = editUsersProfileWi editUsersProfileWi.selectedUser = [selectedUser]

Fig. 11. Solution A: Conditions for opening the edit-profile windows

Solution B. Another solution for the GUI designer consists of specifying, for each of event supported by a widget, *the conditions under which all the actions triggered by the given event can be securely executed by a user*. In our case, the GUI design must convert the edit-profile window model shown in Figure 10 into the one shown in Figure 12.

Widgets	Events		Actions	Arguments
nicknameEn	on-create	if true		
		then	read	object = [selectedUser] attribute = nickname variable = text
passwordEn	on-create	if [role] = 'User' and [caller] = [selectedUser]		
		then	read	object = [selectedUser] attribute = password variable = text
moodMsgEn	on-create	if true		
		then	read	object = [selectedUser] attribute = moodMsg variable = text
emailEn	on-create	if (([role] = 'User' and [caller] = [selectedUser]) or [role] = 'Admin')		
		then	read	object = [selectedUser] attribute = email variable = text
statusEn	on-create	if true		
		then	read	object = [selectedUser] attribute = status variable = text
updateBu	click-on	if [role] = 'User' and [caller] = [selectedUser]		
		then	update	object = [selectedUser] attribute = nickname value = [nicknameEn.text]
			update	object = [selectedUser] attribute = password value = [passwordEn.text]
			update	object = [selectedUser] attribute = moodMsg value = [moodMsgEn.text]
			update	object = [selectedUser] attribute = email value = [emailEn.text]
			update	object = [selectedUser] attribute = status value = [statusEn.text]

Fig. 12. Solution B: The ChitChat edit-profile window (now security-aware)

This solution is certainly simpler and less intrusive than Solution A with respect to the original, security-unaware GUI model (since, for example, no new windows need to be added to the original design). However, in order to implement this solution, the GUI designer must gather all the knowledge corresponding to (ii), in Step 2 above, for every event supported by every widget in the model.

A model-transformation approach. Clearly, manually modeling the application's security policy within the GUI model is problematic. It is cumbersome, error prone, and scales poorly to large applications. Moreover, it requires the GUI designer to have complete knowledge of the access control policy on the application data. Finally, the resulting GUI models are difficult to maintain.

To address the problem of establishing the link between visualization and security, we employ a standard technique from model-driven development: model transformation. A model transformation takes as input a model (or several models) conforming to given metamodel (respectively, several metamodels) and produces as output a model conforming to a given metamodel. More specifically, the ActionGUI Toolkit implements a model transformation that takes as input an ActionGUI model and a SecureUML+ComponentUML model, and *automatically* produces as output an ActionGUI model. This output model is identical to the input ActionGUI model except that it is now security-aware with respect to the access control policy specified in the input SecureUML+ComponentUML model. In particular, our model transformation follows the ideas behind Solution B: it specifies for each event supported by a widget, the conditions under which all the actions triggered by this event can be securely executed by a user. As expected, at the core of this model transformation, we have implemented a function (on the input models) that *automatically* gathers all the knowledge corresponding to (i) and (ii), in Step 1 and Step 2 above, for each action triggered by an event and for each event supported by a widget, respectively.

Continuing with our running example, let us consider how models can be made security-aware and maintained over time. First, how can the GUI designer model the edit-profile window to make it aware of and respect the security requirements (4–6)? Using our ActionGUI Toolkit, the designer simply calls the aforementioned model transformation on ChitChat's access control policy shown in Figure 6 and the (security-unaware) ChitChat edit-profile window model shown in Figure 10. Our model transformation then automatically generates (in practically no time) the security-aware ChitChat edit-profile window model shown in Figure 12. Finally, what must the GUI designer do, with respect to the edit-profile window model, if ChitChat's security policy happens to change? For example, suppose that any user is allowed to read the email of other users when the former participates in a chat room where the latter is also participating. The designer must simply call again the model transformation, this time taking as inputs the modified model of ChitChat's access control policy and, as before, ChitChat's (security-unaware) edit-profile window model. shown in Figure 10.

6 Related Work and Conclusions

The ever-growing development and use of information and communication technology is a constant source of security and reliability problems. Clearly we need better ways of developing software systems and approaching software engineering as a well-founded discipline. In many engineering disciplines, model building is at the heart of system design. This is increasingly the case in software engineering

since model-driven software engineering was first proposed over a decade ago. In our opinion, the late adoption of this methodology is due to the difficulty in defining effective domain-specific modeling languages and also the effort required for developers to learn modeling languages and the art of model building.

Defining a good domain-specific modeling language requires finding the right abstractions and degree of precision to capture relevant aspects of the structure and the logic of a software system. In addition, for a modeling language to be really usable and useful for software developers, appropriate tools must be provided to build models, analyze them, and keep them synchronized with end products. We wish to emphasize in this regard the need to focus on concrete domains like access controls, GUIs, data models, and the like. In our experience, only by limiting the domain is it possible to build sufficiently precise modeling languages that support the automatic generation of fully functional applications.

Automatic code generation brings with it important advantages. Once a code generator is implemented for a platform (a one-off cost), modelers can use it like a compiler for a very high-level language, dramatically increasing their productivity. But even when the model-driven development process is not completely automatic, there are experience reports (see, for instance, [9,11]), in which the productivity of a developer in industry is said to increase by a factor of 2 or 3.

There is an additional argument for using model-driven development to develop security-critical systems, i.e., for model-driven security [1]. Namely, security is often built redundantly into systems. For example, in a web-application, access control may be enforced at all tiers: at the web server, in the back-end databases, and even in the GUI. There are good reasons for this. Redundant security controls is an example of defense in depth and is also necessary to prevent data access in unanticipated ways, for example, directly from the database thereby circumventing the web application server. Note that access control on the client is also important, but more from the usability rather than the security perspective. Namely, although client-side access control may be easy to circumvent, it enhances usability by presenting honest users an appropriate view of their options: unauthorized options can be suppressed and users can be prevented from entering states where they are unauthorized to perform any action, e.g., where their actions will result in security exceptions thrown by the application server or database.

The above raises the following question: must one specify security policies separately for each of these tiers? The answer is “no” for many applications. Security can often be understood in terms of the criticality of data and an access control policy on data can be specified at the level of component (class) models, as discussed in Section 3. Afterwards, an access control policy modeled at the level of components may be lifted to other tiers. When the tiers are also modeled, this lifting can be accomplished using model transformation techniques and in a precise and meaningful way as we have illustrated in Section 5. In general, model transformations support problem decomposition during development where design aspects can be separated into different models which are later composed. As a methodology for designing security-aware GUIs, this approach supports

the consistent propagation of a security policy from component models to GUI models and, via code generation, to GUI implementations. This decomposition also means that security engineers and GUI designers can independently model what they know best and maintain their models independently.

Related Work

There are also other tools like WebRatio [12], Olivanova [4], and Lightswitch [8] that support development methodologies for building data-centric applications that are similar to the model-driven methodology presented in this work. In these tools, application development starts by building a data model that reflects the data structure required for the database. The development process continues by applying different UI generation patterns to the data model. These patterns enable data retrieval, data editing, data creation, and database search. A detailed comparison of the expressiveness provided by the languages supported by these tools with the languages supported by the ActionGUI Toolkit is interesting; however it falls out of the scope of this paper. Nevertheless, we note that, in contrast to what these tools can provide, the ActionGUI Toolkit offers developers the full flexibility to create designs without burdening them with the restrictions imposed by the obligatory use of a fixed number of given patterns. The above tools also impose a major restriction at the level of data management, i.e., at the level of data access and visualization: The information that can be referenced and therefore that can be accessed and visualized within one screen can only come from one table of the database or, at most, from two tables that are reachable from each other within one navigation step.

The three tools, WebRatio, Olivanova, and Lightswitch, support the definition and generation of RBAC policies at different granularity levels. Lightswitch supports granting or denying permissions (whose actual behavior must be manually programmed) to execute create, read, update, or delete actions on entities for users in different roles. WebRatio and Olivanova also support granting or denying permissions to execute a similar set of actions on entity's properties, individually, for users in different roles. In WebRatio and Olivanova, the role of the authorization constraints could be played by preconditions restraining the invocation of actions through a concrete UI. Note, however, that none of these tools implement an algorithm capable of lifting to the UIs the security policy that governs the access to data.

Future Work

The toolkit we presented supports the construction of security, data, and GUI models and generates complete, deployable, security-aware web applications from these models. However, there is still much work ahead to turn this toolkit into a full, robust commercial application. In particular, we plan to add to the language (and to its code-generator) other actions which do not act upon the database or the GUI elements. Examples are sending an email to a contact selected in a list or printing a table.

Our work here, as well as our past work in model-driven security, has focused primarily on access control. However, many systems have security requirements that go beyond access control, for example, obligations on how data must or must not be used once access is granted. We are currently working on handling usage control policies in the context of model-driven security. The challenge here is to define modeling languages that are expressive enough to capture these policies, support their formal analysis, and provide a basis for generating infrastructures to enforce or, at least, monitor these policies.

There are many challenging questions concerning model analysis. Here, our goal is to be able to analyze the consistency of different system views. For example, suppose that access control is implemented at multiple tiers (or levels) of a system, e.g., at the middle tier implementing a controller for a web-based application and at the back-end persistence tier. If the policies for both of these tiers are formally modeled, we would like to answer questions like "will the controller ever enter a state in which the persistence tier throws a security exception?" Note that with advances in model transformations, perhaps such questions will some day not even need to be asked, as we can uniformly map a security policy across models of all tiers.

Ultimately we see model-driven security playing an important role in the construction and certification of critical systems. For example, certification under the Common Criteria requires models for the higher Evaluation Assurance Levels. Model-driven security provides many of the ingredients needed: models with a well-defined semantics, which can be rigorously analyzed and have a clear link to code. As the acceptance of model-driven development techniques spread, and as they become better integrated with well-established formal methods that support a detailed behavioral analysis, such applications should become a reality.

Acknowledgements. This work is partially supported by the EU FP7-ICT Project "NESSoS: Network of Excellence on Engineering Secure Future Internet Software Services and Systems" (256980) by the Spanish Ministry of Science and Innovation Project "DESAFIOS-10" (TIN2009-14599-C03-01), and by Comunidad de Madrid Program "PROMETIDOS-CM" (S2009TIC-1465).

References

1. Basin, D., Clavel, M., Egea, M.: A decade of model driven security. In: Proceedings of the 16th ACM Symposium on Access Control Models and Technologies (SACMAT 2011). ACM Press, New York (2011) (invited paper, in press)
2. Basin, D., Clavel, M., Egea, M., Schläpfer, M.: Automatic generation of smart, security-aware GUI models. In: Massacci, F., Wallach, D., Zannone, N. (eds.) ESSoS 2010. LNCS, vol. 5965, pp. 201–217. Springer, Heidelberg (2010)
3. Basin, D., Doser, J., Lodderstedt, T.: Model driven security: From UML models to access control infrastructures. ACM Transactions on Software Engineering and Methodology 15(1), 39–91 (2006)
4. Care Technologies. Olivanova – the programming machine (2011), `http://www.care-t.com`

5. Egea, M., Dania, C., Clavel, M.: MySQL4OCL: A stored procedure-based MySQL code generator for OCL. Electronic Communications of the EASST 36 (2010)
6. Ferraiolo, D.F., Sandhu, R.S., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed NIST standard for role-based access control. ACM Transactions on Information and System Security 4(3), 224–274 (2001)
7. Kleppe, A., Bast, W., Warmer, J.B., Watson, A.: MDA Explained: The Model Driven Architecture–Practice and Promise. Addison-Wesley, Reading (2003)
8. Microsoft. Visual studio lightswitch (2010), http://www.microsoft.com/visualstudio/en-us/lightswitch
9. Mohan, R., Kulkarni, V.: Model driven development of graphical user interfaces for enterprise business applications - experience, lessons learnt and a way forward. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 307–321. Springer, Heidelberg (2009)
10. Object Management Group. Object Constraint Language specification Version 2.2 (February 2010), OMG document, http://www.omg.org/spec/OCL/2.2
11. Schramm, A., Preußner, A., Heinrich, M., Vogel, L.: Rapid UI development for enterprise applications: Combining manual and model-driven techniques. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 271–285. Springer, Heidelberg (2010)
12. Web Models Company. Web ratio – you think, you get (2010), http://www.webratio.com

SSG: A Model-Based Development Environment for Smart, Security-Aware GUIs

Miguel A. García de Dios
IMDEA Software Institute
miguel.garcia@imdea.org

Carolina Dania
IMDEA Software Institute
carolina.dania@imdea.org

Michael Schläpfer
ETH Zürich
michschl@inf.ethz.ch

David Basin
ETH Zürich
basin@inf.ethz.ch

Manuel Clavel
IMDEA Software Institute
manuel.clavel@imdea.org

Marina Egea
ETH Zürich
marinae@inf.ethz.ch

ABSTRACT

We present a development environment for automatically building smart, security-aware GUIs following a model-based approach. Our environment consists of a number of plugins that have been developed using the Eclipse framework and includes three model editors, a model-transformation tool, and a code generator.

1. INTRODUCTION

In many programs, users access application data using GUI widgets: data is created, deleted, read, and updated using text boxes, check boxes, buttons, and the like. There is an important, but little explored, link between visualization and security. When the application data is protected by an access control policy, the application GUI should be *aware of* and *respect* this policy. For example, the GUI should not display options to users for actions that they are not authorized to execute on application data. This prevents user frustration, for example, from filling out a long electronic form only to have the server reject it because the user lacks permissions to execute some associated actions on the application data. Taking this idea one step further, the GUI should not, for example, display options to users to open other widgets when these widgets only display options for actions that the users are not authorized to execute on application data. That is, the application GUI should not only be security-aware, it should also be *smart*.

Here we present an environment for developing such GUIs, using the Eclipse framework. Our environment supports linking visualization and security during system design and using this design information to automatically generate GUI implementations that are both smart and security-aware. To the best of our knowledge, no other GUI development environment currently provides this kind of support, either for design or implementation.

ICSE '10, May 2-8 2010, Cape Town, South Africa

2. MODEL-BASED DEVELOPMENT OF SMART, SECURITY-AWARE GUIS

The default, ad-hoc approach to linking visualization and security would be to directly hardcode the security policy within the GUI. But this is clearly inadequate. First, the GUI designer is often not aware of the application data security policy. Second, even if the designer is aware of it, hardcoding the application-data security policy within the GUI code is cumbersome and error-prone, if done manually. Finally, any changes in the security policy will require manual changes to the GUI code where this policy is hardcoded, which again is a cumbersome and error-prone task.

In [1] we propose a model-based approach to linking visualization and security. The key idea is that this link is ultimately defined in terms of *data actions*, since data actions are both controlled by the security policy and triggered by the events supported by the graphical user interface. The key component of this solution is a many-models-to-model transformation which, given a security-design model and a GUI model, automatically generates a GUI model that is both security-aware and smart.

Thus, under this model-based development approach, illustrated in Figure 1, the process of building a smart, security-aware GUI has the following parts.

1. Software engineers specify both the application-data model $\mathcal{C}$ and the security-design model $\mathcal{S}_\mathcal{C}$.
2. GUI designers specify the application GUI model $\mathcal{G}_\mathcal{C}$.
3. A many-models-to-model transformation automatically generates a smart, security-aware GUI model $\mathcal{M}_{(\mathcal{G}_\mathcal{C},\mathcal{S}_\mathcal{C})}$ from the security-design model $\mathcal{S}_\mathcal{C}$ and the GUI model $\mathcal{G}_\mathcal{C}$.
4. A code generator automatically produces the smart, security-aware GUI from the smart, security-aware GUI model $\mathcal{M}_{(\mathcal{G}_\mathcal{C},\mathcal{S}_\mathcal{C})}$.

As a design methodology, our model-based approach has three main advantages over traditional approaches to user interface design. First, security engineers and GUI designers can independently model what they know best. Second, security engineers and GUI designers can independently change their models, and these changes are automatically propagated to the security-aware GUI models. Third, GUI designers can use the generated security-aware GUI models to check that they are designing the right GUI to give the (authorized) users access to the (intended) application data.

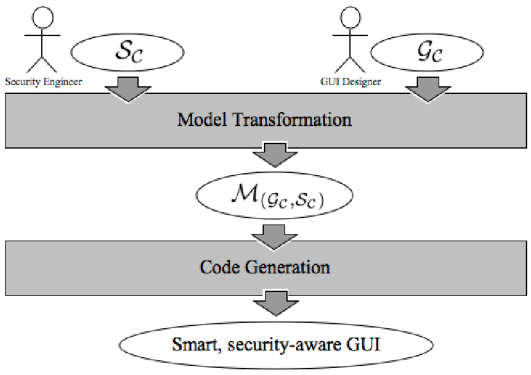


Figure 1: Modeling a smart and security-aware GUI.

3. SSG: A SMART, SECURITY-AWARE GUI BUILDER

SSG is a development environment, built using the Eclipse framework, for generating smart, security-aware GUIs following the model-based approach described in Section 2. In what follows, we describe the plugins included in SSG. All of the plugins are publicly available at [3].

3.1 Data model editor

The data model GMF-editor allows users to graphically model application data. This editor supports a simple language, named ComponentUML, for modeling application data. Essentially, this language provides a subset of UML class models: entities can be related by associations and may have attributes. Hence, the editor provides a concrete graphical syntax for modeling entities, with their attributes and association-ends.

3.2 Security-design model editor

The security-design model GMF-editor allows users to model an application's access control policy. This editor supports a language, named SecureUML+ComponentUML [2], for modeling access control policies on ComponentUML resources, i.e., on entities, their attributes, and associations. The policies that can be specified in SecureUML+ComponentUML are of two kinds: those that depend on static information, namely the assignments of users and permissions to roles, and those that depend on dynamic information. The actions that can be controlled in SecureUML+ComponentUML are, e.g., those to 'create' and 'delete' entities, and to 'read' and 'update' their attributes. SecureUML+ComponentUML also provides composite actions, which group primitive actions into a hierarchy of higher-level ones. The composite actions are 'read', 'update', and 'full access' either on entities or entity's properties. For example, 'full access' on an attribute includes both 'read' and 'update' access on the attribute.

3.3 GUI model editor

The GUI model GMF-editor allows users to model an application's graphical user interface. This editor supports a language, named GUI [1, 3], for modeling the behavioral properties of GUIs, namely what are the actions associated to the different events that are supported by the GUI. In a nutshell, this language can be used to model GUIs that consist of widgets (buttons, entries, labels) that are displayed inside containers (windows, combo-boxes), which are themselves widgets. Each widget has a set of associated events (e.g., on-click and on-create). These are the events supported by the widget. Each event is associated with a set of actions: these are the actions triggered by the event. Events' actions are of two types: widget actions (which are actions on GUI widgets, e.g., open, close, focus, and set) and data actions (which are actions on the application data). Both widget and data actions may take parameters. Also, each container has a (possibly empty) set of variables associated with it: these variables hold information that can be used by actions within this container.

3.4 GUI model generator

This QVT-generator automatically transforms a GUI model and a security-design model (both sharing the same data model) into a model of a new GUI. The resulting model has the same behavioral properties as the one modeled by the given GUI model, except that it is now both smart and security-aware with respect to the access control policy modeled by the given security-design model. The new GUIs are modeled using a language, named SecureUML+GUI, that supports modeling the behavioral properties of GUIs along with the information about which role can execute which events on these GUIs.

3.5 Code generator

The JET-code generator automatically generates, from a smart, security-aware GUI model, a full web application consisting of a collection of PHP-web pages whose design and behavior implements those modeled by the given smart, security-aware GUI model. In particular, windows are implemented as web pages. Thus, opening a window is implemented as loading the corresponding page and closing a window is implemented as loading the previously visited page. More interestingly, data actions (like 'create' or 'delete' entities and 'update' or 'read' their attributes) are implemented as SQL queries or statements on a data-base implementing the underlying data model; for convenience, the code generator can also create this database for the user. Finally, permissions to execute events on widgets (like clicking a button or creating an entry or a text box) are implemented by appropriate conditional statements in the PHP-code responsible for interpreting those events.

4. REFERENCES

[1] D. Basin, M. Clavel, M. Egea, and M. Schläpfer. Automatic generation of smart, security-aware GUI models. In F. Massacci, D. Wallach, and N. Zannone, editors, *Proceedings of the 2nd International Symposium on Engineering Secure Software and Systems (ESSOS 2010)*, volume 5695 of *LNCS*, pages 201–217. Springer-Verlag, 2010.

[2] D. Basin, J. Doser, and T. Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology*, 15(1):39–91, 2006.

[3] B. S. Group. The SmartGUI Project. `http://www.bm1software.com/`, 2009.

Electronic Communications of the EASST
Volume 24 (2009)

Proceedings of the Workshop The Pragmatics of OCL and Other Textual Specification Languages at MoDELS 2009

Checking Unsatisfiability for OCL Constraints

Manuel Clavel, Marina Egea and Miguel A. García de Dios

13 pages

Guest Editors: J. Cabot, J. Chimiak-Opoka, F. Jouault, M. Gogolla, A. Knapp
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Checking Unsatisfiability for OCL Constraints

Manuel Clavel[1], **Marina Egea**[2] **and Miguel A. García de Dios**[3]*

[1] manuel.clavel@imdea.org
[3] miguelangel.garcia@imdea.org
IMDEA Software Institute, Madrid, Spain

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain

[2] marinae@inf.ethz.ch
Information Security / ZISC
ETH Zürich, Switzerland

Abstract: In this paper we propose a mapping from a subset of OCL into first-order logic (FOL) and use this mapping for checking the unsatisfiability of sets of OCL constraints. Although still preliminary work, we argue in this paper that our mapping is both simple, since the resulting FOL sentences closely mirror the original OCL constraints, and practical, since we can use automated reasoning tools, such as automated theorem provers and SMT solvers to automatically check the unsatisfiability of non-trivial sets of OCL constraints.

Keywords: OCL, unsatisfiability, automated deduction, SMT solvers

1 Motivation

The lack of tool support for OCL was pointed out in [CBC05] as a main cause for the limited adoption of the language in industry. Since then, many initiatives have been brought to fruition and their outcomes are available to designers (see [Dem05]). Among the tool categories that have received significant attention are:

- *Parsers*: to check the syntactical well-formedness of an expression: e.g., the Dresden OCL 2.0 parser [Kon05, Sof07].
- *Evaluators*: to obtain the value of an expression within a contextual model: e.g., USE [Dat06], MDT OCL [Hus08], and EOS [CED08, DCE08].
- *Translators*: to map (for different purposes) an expression into a (logically equivalent) expression in other languages and/or formalisms: e.g.,
  - OCL2SQL [Hei06, Sof07] maps OCL constraints into SQL queries;

* Research partially supported by Spanish MEC projects TIN2006-15660-C02-01 and by Comunidad de Madrid Program S-0505/TIC/0407.

- UMLtoCSP [CCR07b, CCR07a] maps OCL constraints into constraint programming expressions to support automated bounded verification of UML class diagrams annotated with OCL constraints;
- MOMENT [CRBG08] and ITP-OCL [CE08] map OCL into equational logic (although using different approaches) to support automated evaluation of OCL expressions using term-rewriting.
- KeY [BS07] includes a mapping of OCL into first-order logic to allow interactive reasoning about UML diagrams with OCL constraints.
- HOL-OCL [Bru07, BW07] maps OCL into higher-order logic also to allow interactive reasoning about UML diagrams with OCL constraints.

The work presented here belongs to the third category: it proposes a mapping from OCL to first-order logic, which is defined with the purpose of supporting (*unbounded*) unsatisfiability checks for OCL expressions using *automated* reasoning tools. In our view, being able to check the *unsatisfiability* of (sets of) OCL expressions is a powerful tool, since it will allow modelers to (among other tasks):

- Verify class invariants, by checking that they logically imply the expected constraints/properties;
- Verify method preconditions, by checking that the class invariants do not logically imply their negations; and
- Verify method postconditions, by checking that they do not logically imply the negation of (any of) the class invariants.

However, also in our view, what will make an unsatisfiability checker not only powerful, but also practical, is being *automated*. Given the undecidable nature of the full OCL language, one can only expect to have an automated unsatisfiability checker for a large class of OCL expressions. In this paper we do not attempt to define how large and/or interesting is the class of unsatisfiable OCL expressions that we are able to check automatically. Nevertheless, we will argue that our mapping is both simple, since the resulting FOL sentences closely mirror the original OCL constraints, and practical, since we can use existing automated theorem provers (e.g., Prover9 [McC06]) and/or SMT solvers (e.g., Yices [DM08]) to automatically check the unsatisfiability of non-trivial sets of OCL constraints.

Organization In Section 2 we define our notion of *unsatisfiability* for OCL constraints. Then, in Section 3 we define our mapping from OCL to FOL. Next, in Section 4 we report on our experience using two different automated reasoning tools for checking the unsatisfiability of (sets of) OCL constraints: namely, Prover9 [McC06] (an automated theorem prover) and Yices [DM08] (an SMT solver). We conclude with a discussion on related and future work.

Figure 1: A simple *Library*-model.

2 Unsatisfiability of OCL constraints

In this section we introduce our notion of *unsatisfiability* for OCL constraints, as well as the examples that we will use in the following sections. The notion of *unsatisfiability* that we propose emphasizes the logical meaning of OCL constraints; in fact, it basically translates to OCL the standard notion of unsatisfiability for logic formulas. There are other notions of satisfiability/unsatisfiability for OCL constraints in the literature, which we will briefly discuss at the end of this section.

In what follows, we denote by OCL *constraint* any OCL expression of type Boolean. We do not assume that instances of models always have a finite a number of elements.

Definition 1 Given a model (class diagram) $\mathscr{M}$, and a set of OCL constraints Φ, we say that Φ is $\mathscr{M}$*-unsatisfiable* if and only if there does not exist an $\mathscr{M}$-instance (object diagram) $\mathscr{O}$ on which every constraint in Φ evaluates to true.

To illustrate this notion, we introduce the following example. Table 1 shows a list of OCL constraints: they all refer to the simple class diagram *Library* shown in Figure 1. In this model, libraries contains *Books* and books have *Authors*, *pages*, and an *ISBN* code.

According to Definition 1, the following subsets (among others) of the constraints shown in Table 1 are *Library*-unsatisfiable: {1,2}, {1,8}, {1,10}, {2,3}, {2,4}, {2,5}, {2,6}, {7,8}, {11, 13}, {12}, {14}, and {15}.

Notice that the subset {9,10,11} is not *Library*-unsatisfiable: a library with just one book will satisfy these constraints. On the other hand, the subset {9,10,11,16} is indeed *Library*-unsatisfiable. Notice also that the subset {17} is not *Library*-unsatisfiable: a library with an infinite number of books will satisfy this constraint. On the other hand, the subset {17, 18} is indeed *Library*- unsatisfiable.

As mentioned before, other notions of satisfiability/unsatisfiability of UML models with OCL constraints can be found in the literature. In particular, the notions used in [CCR07b, CT07] are those of *weak* and *strong satisfiability* (and related notions are also introduced in [BW07]). Weak satisfiability means that there exists a finite instance of the model in which at least one class is populated with at least one element. Strong satisfiability means that there exists a finite instance of the model in which all its classes are populated with at least one element. Notice that, if a set of constraints is unsatisfiable (in our sense), then it can not be weak nor strong satisfiable. On the other hand, if a set of constraints is not weak nor strong satisfiable, it does not imply that is unsatisfiable (in our sense).

1. Book.allInstances()−>isEmpty().
2. Book.allInstances()−>exists(x|x.pages > 300).
3. Book.allInstances()−>forAll(x|x.pages < 300).
4. Book.allInstances()−>select(x|x.pages > 300)−>isEmpty().
5. Book.allInstances()−>reject(x|x.pages <= 300)−>isEmpty().
6. Book.allInstances()−>collect(x|x.pages)−>asSet()−>forAll(i|i < 300).
7. Book.allInstances()−>forAll(x|x.author−>isEmpty()).
8. Author.allInstances()−>exists(a|a.books−>notEmpty()).
9. Book.allInstances()−>forAll(x,y|x<>y implies x.isbn<>y.isbn)).
10. Book.allInstances()−>exists(x|Book.allInstances()−>excluding(x)
    −>forAll(y|y.isbn=x.isbn)).
11. Book.allInstances()−>notEmpty().
12. Book.allInstances()−>exists(x|Book.allInstances()−>excludes(x)).
13. Book.allInstances()−>forAll(x|Book.allInstances()−>excluding(x)−>includes(x)).
14. Book.allInstances()−>exists(x|Book.allInstances()−>excluding(x)−>includes(x)).
15. Book.allInstances()−>collect(x|x.author)−>asSet()−>exists(y|y.books
    −>isEmpty())).
16. Book.allInstances()−>size() > 1.
17. Book.allInstances()−>forAll(b|Book.allInstances()−>exists(x|x.pages > b.pages)).
18. Book.allInstances()−>size() = 2.

Table 1: List of constraints

3 A mapping from OCL to FOL

In this section we define a mapping from a subset of OCL into first-order logic (FOL). Given a set of OCL constraints, our mapping generates a set of FOL formulas such that, if the resulting set is unsatisfiable, then the original set is also unsatisfiable. We will argue in Section 4 that our mapping is both simple and practical. At this preliminary stage, we are not ready, however, to provide a formal proof of the *correctness* of this mapping (with respect to a well-defined formal semantics for OCL).

OCL constraints specify properties that must be satisfied by a model. In order to do so in a concise way, OCL provides different constructors to refer to specific collections of elements. In a nutshell, our mapping is defined recursively over the structure of OCL expressions:

- Boolean-expressions are translated to formulas, which essentially mirror their logical structure; Integer-expressions are basically copied; at this point, we do not consider String-expressions.
- Collection-expressions are translated to predicates, whose meaning is defined by additional formulas generated by the mapping; at this point, we only consider Set-expressions.
- Association-ends are translated by predicates, which are also defined by formulas generated by the mapping; at this point, we do not consider qualified associations.
- Attributes are translated by functions, which are left undefined by the mapping.

The function *map*() below defines our mapping from OCL to FOL. We do not attempt to cover here the full OCL language, but only a subset that is significantly enough so as to show the potential of our proposal.

In what follows, given an iterator variable x, the expression $x^\flat$ denotes a fresh new logical variable. Similarly, given Boolean-expressions *BoolExpr*, object expressions *ObjExpr*, and Set-expressions *SetExpr* and *SetExpr'*, the expressions [collect, *SetExpr*, *SetExpr'*]$^\flat$, [select, *SetExpr*, *BoolExpr*]$^\flat$, [reject, *SetExpr*, *BoolExpr*]$^\flat$, [including, *SetExpr*, *ObjExpr*]$^\flat$, and [excluding, *SetExpr*, *ObjExpr*]$^\flat$ denote fresh new predicate names.

The auxiliary function *name*(), used in the definition of *map*(), is the one in charge of providing unique names for the FOL predicates that translate the different OCL Collection-expressions; it also translates OCL literal values by the corresponding FOL terms.

Definition 2 The auxiliary function name() is defined by the following clauses:

$$
\begin{array}{ll}
\mathrm{name}(\mathit{Integer}) & = \mathit{Integer}. \\
\mathrm{name}(-\mathit{Integer}) & = -\mathit{Integer}. \\
\mathrm{name}(\mathit{Integer}[+ \mid *]\mathit{Integer}') & = \mathit{Integer}[+ \mid \times]\mathit{Integer}'. \\
\mathrm{name}(\mathit{Var}) & = \mathit{Var}. \\
\mathrm{name}(\mathit{ClassId}) & = \mathit{ClassId}. \\
\mathrm{name}(\mathit{ObjExpr}.\mathit{Attr}) & = \mathit{Attr}(\mathrm{name}(\mathit{ObjExpr})). \\
\mathrm{name}(\mathit{ObjExpr}.\mathit{AssocEnd}) & = \mathit{AssocEnd}(\mathrm{name}(\mathit{ObjExpr})). \\
\mathrm{name}(\mathit{ClassExpr}.\mathsf{allInstances}()) & = \mathrm{name}(\mathit{ClassExpr}). \\
\mathrm{name}(\mathit{SetExpr}{->}\mathsf{collect}(x|\mathit{SetExpr}')) & = [\mathsf{collect}, \mathit{SetExpr}, \mathit{SetExpr}']^\flat. \\
\mathrm{name}(\mathit{SetExpr}{->}\mathsf{select}(x|\mathit{BoolExpr})) & = [\mathsf{select}, \mathit{SetExpr}, \mathit{BoolExpr}]^\flat. \\
\mathrm{name}(\mathit{SetExpr}{->}\mathsf{reject}(x|\mathit{BoolExpr})) & = [\mathsf{reject}, \mathit{SetExpr}, \mathit{BoolExpr}]^\flat. \\
\mathrm{name}(\mathit{SetExpr}{->}\mathsf{excluding}(x|\mathit{ObjExpr})) & = [\mathsf{excluding}, \mathit{SetExpr}, \mathit{ObjExpr}]^\flat. \\
\mathrm{name}(\mathit{SetExpr}{->}\mathsf{including}(x|\mathit{ObjExpr})) & = [\mathsf{including}, \mathit{SetExpr}, \mathit{ObjExpr}]^\flat.
\end{array}
$$

The auxiliary function *in_coll*(), also used in the definition of *map*(), basically returns the atomic formula that represents the application of a given predicate to a given number of arguments.

Definition 3 The auxiliary function in_coll() is defined by the following clause:

$$\text{in\_coll}(Name, x) \quad = Name(x).$$

Finally, the auxiliary function *make_conj* returns the conjunction of a given set of formulas.

Definition 4 The auxiliary function make_conj() is defined by the following clauses:

$$\begin{aligned} &make\_conj(\emptyset) && = \top. \\ &make\_conj(\{\phi\}) && = \phi. \\ &make\_conj(\{\phi_1, \ldots, \phi_{n+1}\}) && = \phi_1 \wedge \cdots \wedge \phi_{n+1}. \end{aligned}$$

We are now ready to define our mapping from OCL to FOL. First, the function *map*() generates, by default, the sentences defining the predicates that represent, in our mapping, the association-ends specified in the given model.

Definition 5 Given an association between two classes $Class_1$ and $Class_2$, with association-ends $AssocEnd^{Class_1}$ and $AssocEnd^{Class_2}$, the function map() generates, by default, the following sentences:

$$\begin{aligned} &\forall(x,y)(AssocEnd^{Class_1}(x,y) \Rightarrow Class_1(y)). \\ &\forall(x,y)(AssocEnd^{Class_2}(x,y) \Rightarrow Class_2(y)). \\ &\forall(x,y)(AssocEnd^{Class_1}(x,y) \Leftrightarrow AssocEnd^{Class_2}(y,x)). \end{aligned}$$

Next, we define the mapping from OCL Boolean-expressions to FOL formulas: essentially, we mirror the logical structure of the OCL expressions in the resulting FOL formulas. In particular, we map iterator variables into logical variables, which are existentially or universally quantified depending on the iterator used.

Definition 6 The function *map()* on Boolean-expressions is defined by clauses shown in Figure 2.

Finally, we define the mapping from OCL Collection-expressions to FOL formulas. Recall that collections are represented in our mapping by predicates. The function *map*() defines these predicates by generating the appropriate FOL formulas. Recall also that the only Collection-expressions currently covered by our mapping are the Set-expressions.

Definition 7 The function *map()* on Collection-expressions is defined by clauses shown in Figure 3.

4 Examples

In this section, we argue, using a set of examples, that our mapping is both simple and practical.

$$
\begin{aligned}
&\mathrm{map}(\mathsf{true})\\
&\quad = \{\top\}.\\
&\mathrm{map}(\mathsf{false})\\
&\quad = \{\bot\}.\\
&\mathrm{map}(\mathit{IntExpr}[> \mid < \mid >= \mid <= \mid = \mid <>]\mathit{IntExpr}')\\
&\quad = \{\mathrm{name}(\mathit{IntExpr})[>|<|>=|<=|=|<>]\mathrm{name}(\mathit{IntExpr}')\}.\\
&\mathrm{map}(\mathsf{not}\ \mathit{BoolExpr})\\
&\quad = \{\neg(\mathrm{make\_conj}(\mathrm{map}(\mathit{BoolExpr})))\}.\\
&\mathrm{map}(\mathit{BoolExpr}\ \mathsf{and}\ \mathit{BoolExpr}')\\
&\quad = \{\mathrm{make\_conj}(\mathrm{map}(\mathit{BoolExpr})) \wedge \mathrm{make\_conj}(\mathrm{map}(\mathit{BoolExpr}'))\}.\\
&\mathrm{map}(\mathit{BoolExpr}\ \mathsf{or}\ \mathit{BoolExpr}')\\
&\quad = \{\mathrm{make\_conj}(\mathrm{map}(\mathit{BoolExpr})) \vee \mathrm{make\_conj}(\mathrm{map}(\mathit{BoolExpr}'))\}.\\
&\mathrm{map}(\mathit{BoolExpr}\ \mathsf{implies}\ \mathit{BoolExpr}')\\
&\quad = \{\mathrm{make\_conj}(\mathrm{map}(\mathit{BoolExpr})) \Rightarrow \mathrm{make\_conj}(\mathrm{map}(\mathit{BoolExpr}'))\}.\\
&\mathrm{map}(\mathit{SetExpr}{->}\mathsf{isEmpty()})\\
&\quad = \{\forall(x^\flat)(\neg(\mathrm{in\_coll}(\mathrm{name}(\mathit{SetExpr}), x^\flat)))\} \cup \mathrm{map}(\mathit{SetExpr}).\\
&\mathrm{map}(\mathit{SetExpr}{->}\mathsf{notEmpty()})\\
&\quad = \{\exists(x^\flat)(\mathrm{in\_coll}(\mathrm{name}(\mathit{SetExpr}), x^\flat))\} \cup \mathrm{map}(\mathit{SetExpr}).\\
&\mathrm{map}(\mathit{SetExpr}{->}\mathsf{excludes}(\mathit{ObjExpr}))\\
&\quad = \{\neg(\mathrm{in\_coll}(\mathrm{name}(\mathit{SetExpr}), \mathrm{name}(\mathit{ObjExpr})))\} \cup \mathrm{map}(\mathit{SetExpr}).\\
&\mathrm{map}(\mathit{SetExpr}{->}\mathsf{includes}(\mathit{ObjExpr}))\\
&\quad = \{\mathrm{in\_coll}(\mathrm{name}(\mathit{SetExpr}), \mathrm{name}(\mathit{ObjExpr}))\} \cup \mathrm{map}(\mathit{SetExpr}).\\
&\mathrm{map}(\mathit{SetExpr}{->}\mathsf{exists}(\mathsf{x}|\mathit{BoolExpr}))\\
&\quad = \{\exists(x^\flat)(\mathrm{in\_coll}(\mathrm{name}(\mathit{SetExpr}), x^\flat) \wedge \mathrm{make\_conj}(\mathrm{map}(\mathit{BoolExpr}[x \mapsto x^\flat]))))\}\\
&\qquad \cup \mathrm{map}(\mathit{SetExpr}).\\
&\mathrm{map}(\mathit{SetExpr}{->}\mathsf{forAll}(\mathsf{x}|\mathit{BoolExpr}))\\
&\quad = \{\forall(x^\flat)(\mathrm{in\_coll}(\mathrm{name}(\mathit{SetExpr}), x^\flat) \Rightarrow \mathrm{make\_conj}(\mathrm{map}(\mathit{BoolExpr}[x \mapsto x^\flat]))))\}\\
&\qquad \cup \mathrm{map}(\mathit{SetExpr}).
\end{aligned}
$$

Figure 2: Definition: *map*() on Boolean-expressions.

$$
\begin{aligned}
&\mathrm{map}(\mathit{ClassId}.\mathsf{allInstances}())\\
&\quad = \emptyset.\\
&\mathrm{map}(\mathit{SetExpr}{->}\mathsf{collect}(x|\mathit{SetExpr}'){->}\mathsf{asSet}())\\
&\quad = \{\forall(y^\flat)(\mathrm{in\_coll}(\mathrm{name}(\mathit{SetExpr}{->}\mathsf{collect}(x|\mathit{SetExpr}'),y^\flat)\\
&\qquad \Leftrightarrow \exists(w^\flat)(\mathrm{in\_coll}(\mathrm{name}(\mathit{SetExpr}),w^\flat)\wedge\\
&\qquad\qquad \mathrm{in\_coll}(\mathrm{name}(\mathit{SetExpr}'[x\mapsto w^\flat]),y^\flat)))\}.\\
&\mathrm{map}(\mathit{SetExpr}{->}\mathsf{collect}(x|\mathit{ObjExpr}){->}\mathsf{asSet}())\\
&\quad = \{\forall(y^\flat)(\mathrm{in\_coll}(\mathrm{name}(\mathit{SetExpr}{->}\mathsf{collect}(x|\mathit{ObjExpr})),y^\flat)\\
&\qquad \Leftrightarrow \exists(w^\flat)(\mathrm{in\_coll}(\mathrm{name}(\mathit{SetExpr}),w^\flat)\wedge\\
&\qquad\qquad y^\flat = \mathrm{name}(\mathit{ObjExpr}[x\mapsto w^\flat]))\}.\\
&\mathrm{map}(\mathit{SetExpr}{->}\mathsf{select}(x|\mathit{BoolExpr}))\\
&\quad = \{\forall(y^\flat)(\mathrm{in\_coll}(\mathrm{name}(\mathit{SetExpr}{->}\mathsf{select}(x|\mathit{BoolExpr}[x\mapsto y^\flat])),y^\flat)\\
&\qquad \Leftrightarrow (\mathrm{in\_coll}(\mathit{name}(\mathit{SetExpr}),y^\flat)\wedge\\
&\qquad\quad \mathrm{make\_conj}(\mathrm{map}(\mathit{BoolExpr}[x\mapsto y^\flat])))\}.\\
&\mathrm{map}(\mathit{SetExpr}{->}\mathsf{reject}(x|\mathit{BoolExpr}))\\
&\quad = \{\forall(y^\flat)(\mathrm{in\_coll}(\mathrm{name}(\mathit{SetExpr}{->}\mathsf{reject}(x|\mathit{BoolExpr}[x\mapsto y^\flat])),y^\flat)\\
&\qquad \Leftrightarrow (\mathrm{in\_coll}(\mathrm{name}(\mathit{SetExpr}),y^\flat)\wedge\\
&\qquad\quad \neg(\mathrm{make\_conj}(\mathrm{map}(\mathit{BoolExpr}[x\mapsto y^\flat]))))\}.\\
&\mathrm{map}(\mathit{SetExpr}{->}\mathsf{excluding}(\mathit{ObjExpr}))\\
&\quad = \{\forall(y^\flat)(\mathrm{in\_coll}(\mathrm{name}(\mathit{SetExpr}{->}\mathsf{excluding}(\mathit{ObjExpr})),y^\flat)\\
&\qquad \Leftrightarrow (\mathrm{in\_coll}(\mathrm{name}(\mathit{SetExpr}),y^\flat)\wedge y^\flat \neq \mathrm{name}(\mathit{ObjExpr})))\}.\\
&\mathrm{map}(\mathit{SetExpr}{->}\mathsf{including}(\mathit{ObjExpr}))\\
&\quad = \{\forall(y^\flat)(\mathrm{in\_coll}(\mathrm{name}(\mathit{SetExpr}{->}\mathsf{including}(\mathit{ObjExpr})),y^\flat)\\
&\qquad \Leftrightarrow (\mathrm{in\_coll}(\mathrm{name}(\mathit{SetExpr}),y^\flat)\vee y^\flat = \mathrm{name}(\mathit{ObjExpr})))\}.
\end{aligned}
$$

Figure 3: Definition: *map*() on Collection-expressions.

.

4.1 Mapping constraints

With respect to other mappings previously proposed, one advantage of our mapping is that, in addition to be mechanizable, the resulting formulas are similar to the original OCL constraints, both in their size and in their logical structure. In this sense, we claim that our mapping is *simple*. To illustrate this claim, we show in Figure 4 the FOL formulas that the function *map*() generates to define the predicates that represent, in our mapping, the two association-ends in the *Library*-model. More interestingly, we show in Figure 5 the FOL formulas resulting from applying the function *map*() to a representative subset of the constraints listed in Table 1.

$$\forall(x,y)(\text{books}(x,y) \Rightarrow \text{Book}(y))$$
$$\forall(x,y)(\text{author}(x,y) \Rightarrow \text{Author}(y))$$
$$\forall(x,y)(\text{books}(x,y) \Leftrightarrow \text{author}(y,x))$$

Figure 4: Example: Mapping association-ends.

4.2 Checking unsatisfiability

Another crucial advantage of our mapping is that the resulting formulas can be checked for unsatisfiability using *automated* reasoning tools. To illustrate our point, we have tried to automatically prove the unsatisfiability of different subsets of the constraints shown in Table 1 using Prover9 [McC06] (an automated theorem prover) and Yices [DM08] (an SMT solver). The results are shown in Table 2. Both tools finished their tasks in less than a second, running on a standard laptop computer. In our experiments, we used both tools with their default (command) options.

Prover9 [McC06] is a resolution/paramodulation automated theorem prover for first-order and equational logic. It uses two default limits which, although good in practice, can prevent proofs from being found. Not surprisingly (since it does not support integer arithmetic), Prover9 could not automatically prove the unsatisfiability of some of the subsets of constraints in our experiment.

Yices [DM08] is a high-performance SMT solver that decides the satisfiability of propositional formulas that mix uninterpreted function symbols and equality with interpreted symbols from various theories, in particular for linear real and integer arithmetic, but also for recursive datatypes, tuples, records, lambda expressions and quantifiers among others. Although Yices is not complete when quantifiers are used, it was able to automatically prove the unsatisfiability of all the subsets of constraints in our experiment. This result is certainly encouraging for our purposes; moreover, we expect to take advantage of its decision procedures for recursive datatypes, tuples, and records, in order to prove the unsatisfiability of more complex subsets of OCL constraints.

(1) map(Book.allInstances()−>isEmpty())
$= \{\forall(x)(\neg \text{Book}(x))\}.$

(2) map(Book.allInstances()−>exists(x|x.pages > 300))
$= \{\exists(x)(\text{Book}(x) \wedge (\text{pages}(x) > 300))\}.$

(3) map(Book.allInstances()−>forAll(x | x.pages < 300))
$= \{\forall(x)(\text{Book}(x) \Rightarrow (\text{pages}(x) < 300))\}.$

(4) map(Book.allInstances()−>select(x|x.pages > 300)−>isEmpty())
$= \{\forall(x)\neg(\text{select1}(x)),$
$\forall(x)(\text{select1}(x) \Leftrightarrow (\text{Book}(x) \wedge (\text{pages}(x) > 300)))\}.$

(7) map(Book.allInstances()−>forAll(x|x.author−>isEmpty()))
$= \{\forall(x)(\text{Book}(x) \Rightarrow \forall(y)(\neg(\text{author}(x,y)\}.$

(8) map(Author.allInstances()−>exists(a|a.books−>notEmpty()))
$= \{\exists(a)(\text{Author}(a) \wedge \exists(x)(\text{books}(a,x)))\}.$

(9) map(Book.allInstances()−>forAll(x,y|x<>y implies x.isbn<>y.isbn))
$= \{\forall(x)(\text{Book}(x) \Rightarrow \forall(y)(\text{Book}(y) \Rightarrow (x \neq y \Rightarrow \text{isbn}(x) \neq \text{isbn}(y))))\}.$

(10) map(Book.allInstances()−>exists(x|Book.allInstances()−>excluding(x)
−>forAll(y|y.isbn=x.isbn)))
$= \{\exists(x)(\text{Book}(x) \wedge [\forall(y)(\text{excluding1}(y) \Rightarrow (\text{isbn}(x) = \text{isbn}(y)))$
$\wedge\forall(z)(\text{excluding1}(z) \Leftrightarrow (\text{Book}(z) \wedge z \neq x))])\}.$

(11) map(Book.allInstances()−>notEmpty())
$= \{\exists(x)(\text{Book}(x))\}.$

(12) map(Book.allInstances()−>exists(x|Book.allInstances()−>excludes(x)))
$= \{\exists(x)(\text{Book}(x) \wedge \neg(\text{Book}(x)))\}.$

(13) map(Book.allInstances()−>forAll(x|Book.allInstances()−>excluding(x)
−>includes(x)))
$= \{\forall(x)((\text{Book}(x) \Rightarrow \text{excluding1}(x))$
$\wedge\forall(y)((\text{Book}(y) \wedge y \neq x) \Leftrightarrow \text{excluding1}(y)))\}.$

(14) map(Book.allInstances()−>exists(x|Book.allInstances()−>excluding(x)
−>includes(x)))
$= \{\exists(x)(\text{Book}(x) \wedge \text{excluding1}(x)$
$\wedge(\forall(y)((\text{Book}(y) \wedge y \neq x) \Leftrightarrow \text{excluding1}(y))))\}.$

(15) map(Book.allInstances()−>collect(x|x.author)−>asSet()
−>exists(y|y.books−>isEmpty()))
$= \{\exists(y)(\text{collect1}(y) \wedge \forall(x)(\neg(\text{books}(y,x))),$
$\forall(z)(\text{collect1}(z) \Leftrightarrow \exists(w)(\text{Book}(w) \wedge \text{author}(w,z))))\}.$

Figure 5: Example: Mapping constraints.

5 Related work

As already mentioned, there are other mappings from OCL into different languages and/or formalisms, each one with its own purposes and target reasoning tool, which makes difficult to do a general comparison. We discuss here only those proposals that support (in one way or another) our same objectives, namely, checking unsatisfiability for OCL constraints. In a nutshell, none of these mappings supports both *unbounded* and *automated* unsatisfiability checks for OCL constraints, as our mapping does, for a significantly subset of the language.

	{1,2}	{1,8}	{1,10}	{2,3}	{2,4}	{2,5}	{2,6}	{7,8}	{11,13}	{12}	{14}	{15}
Prover9	√	√	√	•	√	•	•	√	√	√	√	√
Yices	√	√	√	√	√	√	√	√	√	√	√	√

Table 2: Case study: checking unsatisfiability

The *KeY tool* [BS07] is able to generate proof obligations from checking different properties of UML models with OCL constraints (e.g., invariant preservation), using a translation of OCL into first order predicate logic. To the best of our knowledge, the KeY tool translates OCL collection expressions into first order terms, introducing additional axioms in order to constrain the interpretation of the new function symbols which represent the OCL collections [BKS02].[1] Users can then *interact* with the KeY theorem prover to logically reason about their UML models. A limited amount of automated reasoning is provided, which is far from what is currently offered by modern SMT solvers.

HOL-OCL [BW07] is an *interactive* proof environment for OCL. It is implemented as a shallow embedding of OCL into higher-order logic (HOL) within the theorem prover Isabelle. The resulting translations may be hard to understand by standard software engineers. Also, since the amount of automated reasoning which is supported by HOL-OCL is limited, software engineers may find hard to use this tool when reasoning about their UML models.

UMLtoCSP [CCR07b] provides *bounded* automatic verification of UML models annotated with OCL constraints. The users must limit the search space by explicitly indicating the number of objects in each class, the number of links of each association and the possible values of each attribute. When the tool can not find a satisfying instance within the specified search space, this does not means that the property does not hold: it can still hold for values outside the search space (and the user may try to verify the property with wider intervals).

UML2Alloy [Ana, ABGR07] is a front-end that transforms UML diagrams annotated with OCL constraints into the Alloy notation. It translates the model into a Boolean expression, which is then analysed by the SAT solvers implemented within Alloy. As in the case of UMLtoCSP, the domain must be *bounded* by the user before analysing the model.

6 Conclusions and future work

In this paper we have proposed a mapping from a significant subset of OCL into first-order logic (FOL). Although this is still preliminary work, we have argued that our mapping is both simple, since the resulting FOL sentences closely mirror the original OCL constraints, and practical, since we can use automated theorem provers (e.g., Prover9) and/or SMT solvers (e.g., Yices) to automatically check the unsatisfiability of non-trivial sets of OCL constraints.

In the near future, we plan to extend our mapping to deal with larger subsets of OCL. Since this is preliminary work, the list of currently missing features is certainly large: we discuss here

[1] Since the result is often lengthy and hard to read, it was suggested (via some examples) in [BKS02] to use, as an alternative, a predicative translation to first order formulas. Again, to the best of our knowledge, this line of research has not been followed up within the KeY community. Also the examples shown in [BKS02] do not provide enough information so as to understand how this predicative translation will deal with other cases, like collect-expressions, and including|excluding-expressions, or with nested-iterator expressions.

only the most important ones. First, we should be able to deal with bags. Our idea here is to translate Bag-expressions using predicates, as we do for Set-expressions, but with an additional argument indicating the number of occurrences of a given element in the bag. Of course, the proposed mapping for the different Collect-operations will have to be modified accordingly. Second, we should be able to deal with generalizations. The idea here is to add, by default, the expected sentences formalizing that every element in the subclass-collection also belongs to the super-class collection. Third, we should be able to deal with size-expressions. Here, we do not have yet a general solution: dealing with constraints like (16) and (18) in Table 1 and, in general, with constraints that simply restrict the multiplicity of collections, is rather simple; defining a mapping for arbitrary size-expressions appearing anywhere inside constraints requires further investigation. Finally, it would be interesting to define a mapping for general Collection-expressions and for Tuple-expressions, and to explore the capabilities of SMT solvers to automatically check the unsatisfiability of the resulting formulas. In the long term, we should prove, of course, the correctness of our mapping with respect to a formal semantics of the OCL language.

Bibliography

[ABGR07] K. Anastasakis, B. Bordbar, G. Georg, I. Ray. UML2Alloy: A Challenging Model Transformation. In *ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2007)*. LNCS 4735. 2007.

[Ana] K. Anastasakis. UML2Alloy. http://www.cs.bham.ac.uk/~bxb/UML2Alloy.

[BKS02] B. Beckert, U. Keller, P. H. Schmitt. Translating the Object Constraint Language into First-order Predicate Logic. In *Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark*. 2002.

[Bru07] A. D. Brucker. *An Interactive Proof Environment for Object Oriented Specifications*. PhD thesis, ETH Zurich, 2007.

[BW07] A. D. Brucker, B. Wolff. The HOL-OCL tool. http://www.brucker.ch/, 2007. ETH Zurich.

[BS07] B. Beckert, R. Hähnle, P. H. Schmitt (eds.). *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[CBC05] D. Chiorean, M. Bortes, D. Corutiu. Proposals for a Widespread Use of OCL. In *Tool Support for OCL and Related Formalisms - Needs and Trends– MoDELS'05 Conference Workshop*. Pp. 68–82. 2005.

[CCR07a] J. Cabot, R. Clarisó, D. Riera. A tool for the formal verification of UML/OCL models using Constraint Programming. 2007. http://gres.uoc.edu/UMLtoCSP/.

[CCR07b] J. Cabot, R. Clarisó, D. Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *ASE '07: Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering*. ACM, New York, NY, USA, 2007.

[CE08] M. Clavel, M. Egea. The ITP/OCL tool. http://maude.sip.ucm.es/itp/ocl/, 2008.

[CED08] M. Clavel, M. Egea, M. A. G. de Dios. Building an Efficient Component for OCL Evaluation. *ECEASST* 15, 2008.

[CRBG08] J. Carsí, I. Ramos, A. Boronat, A. Gómez. The MOMENT: MOdel ManageMENT Framework Project. http://moment.dsic.upv.es, 2008.

[CT07] J. Cabot, E. Teniente. Transformation techniques for OCL constraints. *Science Computer Programming* 68(3):152–168, 2007.

[Dat06] Database Systems Group. The UML Specification Environment (USE) tool. 2006. http://www.db.informatik.uni-bremen.de/projects/USE/.

[DCE08] M. A. G. de Dios, M. Clavel, M. Egea. The Eye OCL Software (EOS). 2008. http://maude.sip.ucm.es/eos.

[Dem05] B. Demuth. The OCL Portal. http://www-st.inf.tu-dresden.de/ocl/, 2005.

[DM08] B. Dutertre, L. Moura. Yices: An SMT Solver. http://yices.csl.sri.com/, 2008.

[Hei06] F. Heidenreich. OCL-Codegenerierung für Deklarative Sprachen. Master's thesis, University of Dresden, March 2006. http://dresden-ocl.sourceforge.net.

[Hus08] K. Hussey. MDT-OCL. http://www.eclipse.org, 2008.

[Kon05] A. Konermann. The Parser Subsystem of the Dresden OCL 2.0 Toolkit - Design and Implementation. http://dresden-ocl.sourceforge.net/publications.html, 2005.

[McC06] W. McCune. Prover9. http://www.cs.unm.edu/~mccune/prover9/manual/June-2006C/, 2006.

[Sof07] Software Technology Group. The OCL 2.0 Dresden Toolkit. http://sourceforge.net, 2007.

Electronic Communications of the EASST
Volume 15 (2008)

Proceedings of the 8th International Workshop on OCL Concepts and Tools (OCL 2008) at MoDELS 2008

Building an Efficient Component for OCL Evaluation

Manuel Clavel[1 3], Marina Egea[2], Miguel A. García de Dios[3]

11 pages

Guest Editors: Jordi Cabot, Martin Gogolla, Pieter Van Gorp
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Building an Efficient Component for OCL Evaluation

Manuel Clavel[1] [3], **Marina Egea**[2], **Miguel A. García de Dios**[3]

[1] manuel.clavel@imdea.org
IMDEA Software Institute, Madrid, Spain,
[2] marinae@inf.ethz.ch
Information Security Group
ETH Zurich, Switzerland
[3] clavel@sip.ucm.es miguelgd@fdi.ucm.es
Departamento de Sistemas Informáticos y Computación
Universidad Complutense, Madrid, Spain

Abstract: In this paper we report on our experience developing the Eye OCL Software (EOS) evaluator, a Java component for efficient OCL evaluation. We first motivate the need for an efficient implementation of OCL in order to cope with novel usages of the language. We then discuss some aspects that, based on our experience, should be taken into account when building an OCL evaluator for medium-large scenarios. Finally, we explore various approaches for evaluating OCL expressions on really large scenarios.

Keywords: OCL,evaluation,efficiency,benchmark

1 Motivation

In the recent past we have worked on the definition of a formal and executable semantics for the Object Constraint Language (OCL) [OCL06]. The results of this research appeared in a doctoral dissertation [Ege08] and provide the foundations of the ITP/OCL tool [CE06], a rewriting-based evaluator for OCL expressions on instances of user-defined models. As part of our research, we have looked at different usages of OCL beyond its "initial requirements as a precise modeling language complementing UML specifications." Two related applications have drawn our interest [eA01, BA03a, BA03b, CET07, BMDE08], both having to do with using OCL to analyze user-defined models by evaluating queries on the corresponding instances of their metamodels. Since these instances typically contain a large number of elements, evaluating expressions on them comes at a high computational cost.

Consider, for example, the use of OCL to express metrics for Java programs (this application was suggested to us by members of the Triskell group at IRISA, France). The scenarios on which the program metrics will be evaluated are the instances of the Java metamodel corresponding to the programs: thus, the larger the programs the larger the scenarios[1] and, consequently, the higher the computational cost of evaluating the program metrics.

[1] An an example, the SpoonEMF application, developed by the Triskell group generates, for a standard Java program with 10 lines, a scenario with 113 objects; for a program with 100 lines, one with 1180 objects; and for a program with 500 lines, one with 3470 objects.

We report here on our experience developing the Eye OCL Software [DCE08] (EOS) component, an OCL evaluator designed with the goal of performing efficient evaluation of OCL expressions on medium-large size scenarios. In particular, we discuss i) the need for an efficient implementation on OCL in order to cope with the novel usages of the language; ii) the aspects that we have taken into consideration to improve the efficiency of the EOS evaluator on medium-large scenarios; iii) the limits of the current OCL implementations for dealing with really large scenarios. Although we include the results of applying a benchmark to several OCL evaluators, this paper is not a comparative study (see [GKB08b] for a recent study of this kind). In fact, the results are included here only to show the current performance of some OCL tools on medium-large scenarios and to illustrate the aspects that we consider that should be taken into account when implementing an efficient OCL evaluator for medium-large scenarios. Interestingly, this quality —OCL engine efficiency on medium-large scenarios— is not covered by the benchmark proposed in [GKB08b]: in fact, the largest scenarios scenario proposed for testing OCL engine efficiency in [GKB08a] contains only 42 objects and 42 links among them. Furthermore, despite the results of our benchmark, this paper is not a promotional brochure for our EOS component: as a "product", our OCL evaluator is still in its infancy; the fact is that we have only worked a few months in its implementation, which is rather straightforward except for those aspects that are explicitely discussed in this paper.

To motivate the need for OCL engines that can efficiently evaluate expressions on medium-large size scenarios, we show in Table 1 the time that currently takes to evaluate two given OCL expressions on three different, small-medium size scenarios for a number of OCL evaluators: namely, USE 2.4.0 [Gro06], RoclET [BM07], OCLE [CBC+05], MDT OCL [HT08], and EOS [DCE08].[2] The tests were run on a laptop computer, with Windows XP Professional installed, a processor Intel Pentium M 2.00GHz 600MHz, and 1GB of RAM. Also, in the case of the EOS and USE evaluators, we run the JVM with its parameters `-Xms` and `-Xmx` set to 1024m.

The scenarios considered in these tests are instances of the model Library shown in Figure 1: each scenario is referenced by a number n, which also indicates its "size"; more precisely, for each n, the scenario #n is a library that contains exactly 10^n books, each book with a unique title different from "Hobbit". The OCL expressions used in these tests are

$$\text{Book.allInstances()}->\text{forAll(b}|\text{b.title} <> \text{'Hobbit')} \quad (1)$$

$$\text{Book.allInstances()}->\text{forAll(b1,b2}|\text{b1} <> \text{b2 implies b1.title} <> \text{b2.title}). \quad (2)$$

The first expression says that the library does not contain any book titled "Hobbit", while the second one says that the library does not contain two different books with the same title. Obviously, the cost of evaluating these expressions depends on the number of books in the library and the cost of accessing, and storing for later use, information about these books. For example, in order to evaluate the expressions (1) and (2) on scenario #3 we have to perform, respectively, 10^3 and $2 \times 10^3 \times 10^3$ times the operations of accessing and storing a book's title. But this is

[2] In the case of USE, we have run our experiments using its version 2.4.0. For MDT OCL, we have run the experiments using the plug-in "OCL Interpreter", version 1.2.0v200805130238. Finally, for our experiments in OCLE, we have used its version 2.0.

Scenario	*Expression*	RoclET	OCLE	MDT OCL	EOS	USE
#2	(1)	< 1s	< 1s	< 1s	0ms	230ms
	(2)	> 10m	≈ 4s	< 1s	50ms	240ms
#3	(1)	–	≈ 3s	< 1s	10ms	240ms
	(2)	–	> 10m	≈ 4s	841ms	1s342ms
#4	(1)	–	–	< 1s	20ms	261ms
	(2)	–	–	≈ 5m12s	1m18s	2m12s

Table 1: Evaluating performance on medium-large scenarios.

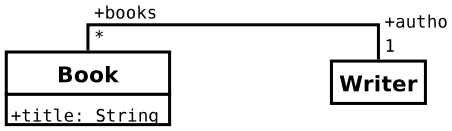


Figure 1: The model Library.

precisely one of the challenges for OCL engines when evaluating expressions on medium-large size scenarios: namely, to efficiently access the information contained in, possibly, all the objects that populate the scenarios.

Notice that in Table 1 we use $> t$ to indicate that we stopped an experiment after having passed time t without obtaining the result. Also, we use a dash to indicate that we did not run the experiment, because we already stopped the execution of a "simpler" one. Finally, since RoclET, OCLE and MDT OCL do not report their execution time in milliseconds, we use < 1s to indicate that the result of an evaluation was output in less than 1 second, and we use $\approx t$ to indicate that the result of an evaluation was output approximately in time t.

2 Measuring the cost of evaluating expressions

Although the computational cost of evaluating a particular OCL expression on a given scenario obviously depends on the algorithms and data structures used to implement each tool, based on our experience, there are two measurements worthwhile considering before launching the evaluation process: first, the maximum number of times that objects' properties will be accessed and, second, the maximum size of the collections that will be built. In the case of medium-large size scenarios, the challenge for OCL engines is that these measurements typically return large numbers.

To illustrate this challenge, we show in Table 2 the performance of MDT OCL, EOS, and USE when evaluating different iterator-expressions whose evaluation require accessing many times objects' properties and/or building large collections. All the expressions in Table 2 were evaluated on the same scenario, namely, an instance MyLibrary of the model Library shown in Figure 1 which contains 10^3 authors, each author with 10 different books, and each book with a title different from "Hobbit".[3] For the sake of the experiment, we artificially increased the

[3] In the case of MDT OCL, the scenario MyLibrary was loaded from an XMI file; for EOS, it was built using a

size of the collections to be iterated upon: in particular, in Table 2, the terms p_1, p_2, and p_3 refer, respectively, to the expressions "Book.allInstances().author.books", "p_1.author.books", and "p_2.author.books", which on the scenario MyLibrary evaluate to collections with 10^5, 10^6, and 10^7 books, respectively.[4] The iterator-expressions in Table 2 were evaluated on a laptop computer, with Windows XP Professional installed, a processor Intel Pentium M 2.00GHz 600MHz, and 1GB of RAM. Also, in the case of the EOS and USE evaluators, we run the JVM with its parameters `-Xms` and `-Xmx` set to 1024m. For the reason explained above, in the case of the MDT OCL evaluator all times shown in Table 2 are approximated ($\approx$).[5] The *Error* evaluation time indicates that we could not evaluate the expression due to an OutOfMemoryError in Java.

Let $exp(p_i)$, with $i \in \{1,2,3\}$, be the time that takes to evaluate an iterator-expression exp on the collection generated by evaluating the term p_i. With respect to the performance of MDT OCL, EOS, and USE, Table 2 shows that, for the same exp, the time $exp(p_{i+1})$ is approximately $10 \times exp(p_i)$ in these tools. Table 2 also shows that, for the same p_i, the time $exp(p_i)$ depends for these tools on the number of accesses to objects' properties that are required to evaluate the body of the iterator-expression exp; we have organized accordingly the expressions in three groups: A, B, and C. Consider, for example, the evaluation times for the first expression in each group. Finally, Table 2 shows that, again for the same p_i, the time $exp(p_i)$ depends as well for these tools on the size of the collection that need to be built. built. Consider, for example, the evaluation time for the first two expressions in Group C: notice that to evaluate

$$p_3 -> \text{collect(x|x.author.books.title)} -> \text{size()} \tag{3}$$

will require to allocate memory for storing a collection with 10^8 titles while evaluating

$$p_3 -> \text{collect(x|x.author.books.title} -> \text{size())} -> \text{sum()} \tag{4}$$

will *only* require to allocate memory for storing a collection with 10^7 integers. Using these and similar expressions, we regularly use the above mentioned measurements, namely, the maximum number of times that objects' properties will be accessed and the maximum size of the collections that will be built, to check the performance of the EOS evaluator and look for possible optimizations.

3 The implementation of the EOS evaluator

As mentioned before, based on our executable equational semantics for OCL [Ege08], we have implemented a rewriting-based OCL evaluator, named ITP/OCL [CE06]. Although our tool

Java program that simply calls the appropriate EOS's interface methods for defining the scenario; and for USE, it was loaded from a file that contained the appropriate USE commands to build the scenario.

[4] A more "natural" approach will be to consider scenarios with larger number of books but, as we will discuss in Section 4, none of the tools support well the loading of scenarios with more than 10^6 objects.

[5] We have also run the same experiments using a desktop computer, with Window Vista Business installed, a processor Intel Core 2 Quad CPU Q9300 2.50GHz 2.50 GHz, and 2GB of RAM. Again, in the case of the EOS and USE evaluators, we run the JVM with its parameters `-Xms` and `-Xmx` set to 1024m. In the case of EOS and MDT OCL the results were similar to those shown in Table 2. In the case of USE, however, the evaluations were completed, approximately, in half the time taken by the single-core test machine.

		MDT	EOS	USE
p_1	->size()	< 1s	30ms	1s12ms
p_2		< 1s	190ms	7s330ms
p_3		≈ 5s	931ms	1m22s
Group A		MDT	EOS	USE
p_1	->collect(x|x.title)->size()	< 1s	80ms	1s212ms
p_2		≈ 1s	391ms	10s84ms
p_3		≈ 18s	3s 896ms	1m48s
p_1	->collect(x|x.title <> 'Hobbit')->size()	< 1s	90ms	981ms
p_2		≈ 2s	481ms	8s432ms
p_3		≈ 20s	4s 516ms	1m30s
Group B		MDT	EOS	USE
p_1	->collect(x|x.author.books)->size()	< 1s	240ms	6s810ms
p_2		≈ 6s	2s 140ms	1m42s
p_3		≈ 58s	17s 736ms	*Error*
p_1	->collect(x|x.author.books->includes(x))->size()	< 1s	221ms	3s565ms
p_2		≈ 7s	1s 893ms	32s677ms
p_3		≈ 1m 1s	17s 906ms	5m32s
p_1	->forAll(x|x.author.books->includes(x))	< 1s	251ms	3s475ms
p_2		≈ 5s	1s 963ms	32s6ms
p_3		≈ 53s	17s 30ms	5m25s
p_1	->select(x|x.author.books->includes(x))->size()	< 1s	260ms	3s685ms
p_2		≈ 5s	2s 13ms	35s411ms
p_3		≈ 55s	17s 605ms	5m51s
Group C		MDT	EOS	USE
p_1	->collect(x|x.author.books.title)->size()	≈ 2s	290ms	8s412ms
p_2		≈ 20s	2s 573ms	1m27s
p_3		≈ 3m 17s	23s 684ms	*Error*
p_1	->collect(x|x.author.books.title->size())->sum()	≈ 2s	270ms	4s957ms
p_2		≈ 19*s*	2s 274ms	48s660ms
p_3		≈ 3m 15s	20s 840ms	10m54s
p_1	->forAll(x|x.author.books.title->excludes('Hobbit'))	≈ 2s	280ms	4s777ms
p_2		≈ 20s	2s 604ms	46s286ms
p_3		≈ 3m 15s	22s 802ms	7m52s

Table 2: Evaluating performance: MDT OCL, EOS, and USE.

	MDT OCL			EOS		USE	
Scenario	XMI	Mem	Time	Mem	Time	Mem	Time
#3	50KB	111MB	$<$ 1s	20MB	40ms	88MB	1s
#4	500KB	112MB	$<$ 1s	22MB	661ms	116MB	35s
#5	5MB	125MB	$\approx$ 45s	50MB	2m36s	209MB	8m25s
#6	50MB		$>$ 20m		$>$ 20m		$>$ 20m

Table 3: Evaluating loading cost: MDT OCL, EOS, and USE.

performs reasonably well on small-medium size scenarios, its performance does not scale up to medium-large size scenarios. Prompted by our interest on applications that require efficient OCL evaluation on medium-large size scenarios, we decided to implement the Eye OCL Software (EOS) evaluator, a Java component whose designed follows the key ideas behind the ITP/OCL tool.

The implementation of the EOS component has taken 4 man-months. It includes an OCL parser (which uses SableCC) and an OCL evaluator, the latter consisting of about 7K lines of Java code. The current version handles most of OCL, including the possibility of adding user-defined operations. With the idea of making it as 'pluggable' as possible, the EOS component is not based on any particular (meta)modeling framework: its public interface provides methods to insert elements, one-by-one, into user-models and scenarios, and to input the expressions to be evaluated as strings of ASCII characters. This decision allowed us also to design the EOS's data structure for internally storing user-models and scenarios in such a way that objects' properties are efficiently accessed. The other possible novelty in its implementation is that, before evaluating a collect expression, we try to (over)estimate the size of the resulting collection and allocate memory in advance. The rest of the EOS implementation is rather straightforward: OCL iterator-expressions are executed using Java for/while loops and standard OCL operations, when possible, are executed using the appropriate Java operators. As expected, expressions are evaluated in EOS following an eager strategy: in particular, collection-expressions are fully evaluated and their resulting elements are all allocated in memory. As part of our research agenda, we plan to study the advantages/disadvantages of a lazy strategy for OCL evaluation, where collection-expressions are only evaluated on-demand.

4 Dealing with really large scenarios

To evaluate expressions on really large scenarios, we need first to solve the problem of loading the scenarios in the OCL evaluators. To illustrate this challenge, we show in Table 3 the time and memory taken by MDT OCL, EOS, and USE, when loading different scenarios of the model Library. Each of the scenarios is identified by a number n, which also indicates its "size": more precisely, for each n, the scenario #n exactly contains 10^n books. Notice that for scenario #6, with 10^6 books, none of the tools were able to finish in less than 20 minutes.[6]

[6] With respect to the "extra" time taken by the EOS tool to store scenarios, compared to MDT OCL, it is possibly due to the extra computation required to store scenarios in the EOS internal data structure.

Scenario	Time
# 1	$\approx$ 0m25s
# 2	$\approx$ 45m

Table 4: Evaluating performance: OCL2SQL

So far, we have explored two different approaches for addressing this problem, both based on the representation of user-models and scenarios as relational databases. The first approach consists on modifying the EOS evaluator so as to look for the information contained in the scenarios directly in its database representation. The advantage of this approach is that it only requires modifying the evaluation of dot-expressions in the expected way: namely, accessing the value of an object's attribute or the value of an object's association-end will be now implemented as a basic SQL `select`-query. The concrete form of these queries depends, of course, on the mapping used to represent models as relational databases (see, for example, [SC08, SI05] and [Gor05]). To check the feasibility of this approach, we modified the EOS evaluator accordingly: unfortunately, the cost of evaluating dot-expressions through the JDBC driver was so high that it made impractical the use of the modified EOS evaluator for evaluating expressions on medium-large scenarios.[7]

The second approach consists on translating OCL expressions into expressions in a query language already available for relational databases. To the best of our knowledge, the most interesting results in this line are discussed in [DHL01, HWD07] and provide the foundations of the OCL2SQL tool [Hei06, Gro07]. However, the solution offered in [DHL01, HWD07] is not satisfactory yet. First, it only considers a restricted subset of the OCL language: in particular, it cannot deal with tuples or nested collections. Second, it only applies to boolean expressions and not to arbitrary queries. Finally, the "complexity" of the SQL expressions resulting from this translation is so high that makes also impractical its use for evaluating expressions on medium-large scenarios.[8] For example, consider a simple extension of model Library shown in Figure 1 where books have now an additional attribute pages. Then, consider the following invariant:

$$\textbf{context}\ \text{Writer}\ \textbf{inv}\text{: self.books}{->}\text{forAll(x} \mid \text{x.pages} > 300) \tag{5}$$

Applying to (5) the translation implemented in OCL2SQL, we automatically obtain the SQL query shown in Figure 2. In Table 4 we show the results of evaluating this query on two different scenarios: scenario #1 contains 10^2 writers and 10^4 books, each writer being the author of 10^2 different books; and scenario #2 contains 10^2 writers and 10^5 books, each writer being the author of 10^3 different books. In both scenarios, all books have exactly 150 pages. The figures correspond to the local execution of the above query in a PostgreSQL 8.3 database installed in a laptop computer, with Windows XP Professional, a processor Intel Pentium M 2.00GHz 600MHz, and 1GB of RAM.

[7] For this experiment, we mapped each class to a table whose columns correspond to its attributes, and each association to a table whose columns correspond to its association-ends.

[8] The OCL2SQL's main developer has confirmed that the efficiency of the OCL2SQL tool has not been tested on medium-large scenarios (e-mail communication, May 2008).

```
create or replace view NAME  as
  (select *
   from OV_Writer as SELF
   where not (not exists (
         select PK_Book
         from (select PK_Book
               from OV_Book as foo
               where PK_Book in
                     (select FK_books
                      from ASS_Ownership as foo
                      where FK_author in
                            (select PK_Writer
                             from OV_Writer as foo
                             where PK_Writer = SELF.PK_Writer)))
        as foo
        where PK_Book in (
              select PK_Book
              from OV_Book as ALIAS2
              where not (ALIAS2.pages> 300)
        )
        )))
```

Figure 2: An example of an SQL query generated by OCL2SQL.

5 Conclusions

In this paper we have first motivated the need for efficient OCL evaluation support in order to cope with the high-computational cost of evaluating OCL expressions in medium-large size scenarios. Then, we have discussed, based on a number of experiments, two measurements that should be taken into consideration when building OCL evaluators for medium-large scenarios: namely, the cost of accessing individual objects' properties and the cost of building collections to hold the partial results of an evaluation. Independently of the results of our benchmark, our aim here is similar to that of [GKB08b]: we do not want to recommend the use of a particular tool, but would like to emphasize the need for a benchmark for OCL engine efficiency on *medium-large* scenarios which can help to build OCL implementations. Next, we have briefly discussed the implementation of the EOS evaluator [DCE08]. Finally, we have presented the challenge of evaluating expressions on really large scenarios and discussed the feasibility of various approaches to address this problem.

Acknowledgements: Research supported by Spanish MEC Project TIN2006-15660-C02-01 and by Comunidad de Madrid Program S-0505/TIC/0407.

Bibliography

[eA01] F. B. e Abreu. Using OCL to formalize object oriented metrics definitions. Technical report ES007/2001, FTC/UNL and INESC, June 2001.
http://ctp.di.fct.unl.pt/QUASAR/Resources/Papers/others/MOOD_OCL.pdf

[BA03a] A. Baroni, F. B. e Abreu. An OCL-based formalization of the MOOSE metric suite. In *ECOOP Workshop on Quantitative Approaches in Object Oriented Software Engineering*. Darmstadt, Germany, July 2003.

[BA03b] A. L. Baroni, F. B. e Abreu. A Formal Library for Aiding Metrics Extraction. In *International Workshop on Object-Oriented Re-Engineering at ECOOP'2003*. Darmstadt, Germany, July 2003.

[BM07] T. Baar, S. Markovic. The RoclET tool. 2007.
http://www.roclet.org/index.php

[BMDE08] D. Basin, M.Clavel, J. Doser, M. Egea. Automated Analysis of Security-Design Models. *Information and Software Technology* 4853, 2008. To appear in the special issue on *Model Based Development for Secure Information Systems*.

[CBC+05] D. Chiorean, M. Bortes, D. Corutiu, C. Botiza, A. Carcu. An OCL Environment (OCLE) 2.0.4. 2005.
http://lci.cs.ubbcluj.ro/ocle/

[CE06] M. Clavel, M. Egea. ITP/OCL: A Rewriting-Based Validation Tool for UML+OCL Static Class Diagrams. In *AMAST'06: 11th International Conference on Algebraic*

Methodology and Software Technology. Lecture Notes in Computer Science 4019, pp. 368–373. Springer, Kuressaare, Estonia, July 2006.

[CET07] M. Clavel, M. Egea, V. Torres. Model Metrication in MOVA: A Metamodel Based Approach using OCL. 2007. Universidad Complutense de Madrid, Spain. http://maude.sip.ucm.es/~marina/pubs/pubs.html

[DCE08] M. A. G. de Dios, M. Clavel, M. Egea. The Eye OCL Software (EOS). 2008. http://maude.sip.ucm.es/eos

[DHL01] B. Demuth, H. Hussmann, S. Loecher. OCL as a Specification Language for Business Rules in Database Applications. In Gogolla and Kobryn (eds.), *UML 2001: 4th International Conference on the Unified Modeling Language*. Lecture Notes in Computer Science 2185, pp. 104–117. Springer, 2001.

[Ege08] M. Egea. *An Executable Formal Semantics for OCL with Applications to Formal Analysis and Validation*. PhD thesis, Universidad Complutense de Madrid, 2008. http://maude.sip.ucm.es/~marina/

[GKB08a] M. Gogolla, M. Kuhlmann, F. Büttner. Sources for a Benchmark for OCL Engine Accuracy, Determinateness, and Efficiency. 2008. http://www.db.informatik.uni-bremen.de/publications/Gogolla_2008_BMSOURCES.pdf/

[GKB08b] M. Gogolla, M. Kuhlmann, F. Büttner. A Benchmark for OCL Engine Accuracy, Determinateness, and Efficiency. In Czarnecki et al. (eds.), *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*. Lecture Notes in Computer Science 5301, pp. 446–459. Springer, 2008.

[Gor05] D. Gornik. A UML Data Modeling profile. February 2005. http://www.jeckle.de/files/RationalUML-RDB-Profile.pdf

[Gro06] D. S. Group. The UML Specification Environment (USE) tool. 2006. http://www.db.informatik.uni-bremen.de/projects/USE/

[Gro07] S. T. Group. The OCL2 Dresden Toolkit. 2007. http://sourceforge.net/project/showfiles.php?group_id=5840

[Hei06] F. Heidenreich. OCL-Codegenerierung für deklarative Sprachen. Master's thesis, University of Dresden, March 2006. http://dresden-ocl.sourceforge.net/publications.html

[HT08] K. Hussey, Model Development Tools Team. MDT-OCL. 2008. http://www.eclipse.org/modeling/mdt/?project=ocl

[HWD07] F. Heidenreich, C. Wende, B. Demuth. A Framework for Generating Query Language Code from OCL Invariants. In *7th OCL Workshop at the UML/MoDELS Conference*. 2007.

[OCL06] Object Management Group. Object Constraint Language specification. May 2006. http://www.omg.org/docs/ptc/05-06-06.pdf

[SC08] S. Sambasivam, P. Crefcoeur. How databases and XML can be used to master UML models, an investigation. *J. Comput. Small Coll.* 23(6):220–228, 2008.

[SI05] Y. Shuxin, R. Indrakshi. Relational Database Operations Modeling with UML. In *AINA '05: Proceedings of the 19th International Conference on Advanced Information Networking and Applications*. Pp. 927–932. IEEE Computer Society, 2005.